Vol.2016-OS-136 No.5
2016/2/29

Recursive Incremental Computation for Efficient Window Aggregate over Array Database

Li Jiang[1,3,a)] Hideyuki Kawashima[2,3] Osamu Tatebe[2,3]

Abstract: An array database is effective for managing and analyzing multi-dimensional scientific big data, and the window aggregate is an important operator in array databases. This paper proposes an efficient method that exploits the scheme of incremental computation and accelerates the execution of window aggregate considerably. Six types of aggregates are improved using different design of buffer tools to eliminate redundant computation. Our proposed recursive incremental computation method completely eliminates all redundant computation. Proposed method is fully implemented in SciDB. Evaluation is conducted on real scientific data as NASA MODIS data. The proposed method achieves performance improvements of 10x for the real application in earth science, comparing with SciDB's built-in window operator. The results align with our time-complexity analysis results.

1. Introduction

Nowadays, scientific fields are growing increasingly data-intensive, efficient tool to manage and analyze big data is getting more and more important. Generally, science data often has multiple dimensions and does not fit into a traditional relational data model easily. For example, spatio-temporal data comes from satellite imagery is very common in several fields, such as earth science, meteorology and geography. Also, telescope images and array sensing data also have at least 3 dimensions. To manage such array data, adoption of the relation data model remains difficult. In theory, the relational database can store multi-dimensional arrays with n-ary relations. However, it incurs a high cost in analytical tasks and in data management on account of impedance mismatching [1].

To efficiently store and analyze such multi-dimensional data, array database systems, such as SciDB [2], [3], [4], SciQL [5], [6], and RasDaMan [7], have been studied. These systems adopt an array model as the basic data model to overcome the impedance mismatch problem. Array database systems have been adopted in some scientific applications that process multi-dimensional arrays, such as in cosmology [8], earth science [9], [10] and experimental physics [11].

One of the important array-oriented queries in such array database system is called **window aggregate**, which is the target query we accelerate in this work. A window aggregate operator executes aggregate functions over a sliding window. Here a window is like a sub-array of the input array with window area sizes in each dimension specified by users. Every cell in the input array has a corresponding window that needs to be computed and the aggregate results are stored in the result array which has the same dimension sizes as the input array. Similar operator can be found in other database systems, but window aggregate operator in array databases has this important characteristic that it is array-oriented, as the data is multi-dimensional and so are the windows to compute in the query.

Window aggregate query has many applications in scientific fields. It is widely used in the raw data cooking process and other analysis tasks. In earth science field, it is often used in order to achieve proper resolution of other analysis results for visualizing purpose [9]. In the field of meteorology, certain analysis tasks involve computation of window aggregate queries [12] in order to forecast weather events. However, in current array databases, straightforward method is used to compute this important operator, which is time-consuming because considerable amount of redundant computation exists during the process.

In this paper, we address the acceleration of window aggregates in array databases. We adopt the scheme of incremental computation to reduce the unnecessary calculation exists in the naive method. The central idea is to buffer the intermediate aggregate results of previously calculated windows and reuse them when computing a new window. We refer the proposed method to as "recursive incremental computation"(RIC) method, which improves the performance with a tradeoff on space for time and it eliminates all redundant computation completely and thus greatly reduce the time cost for the window aggregate tasks. We exploit the plugin mechanism of SciDB to implement the proposed method into the system. The implementation of the recursive IC method demonstrates better efficiency performance compared with the built-in window aggregate operator in SciDB. Our source code is available on GitHub [13], and, all SciDB users can employ it and the scientific community can use it for data processing

[1] Graduate School of Systems and Information Engineering, University of Tsukuba
[2] Faculty of Engineering, Information and Systems, University of Tsukuba
[3] CREST JST CREST
[a)] ljiang@hpcs.cs.tsukuba.ac.jp

tasks.

Several database studies are related to our work. Examples include temporal aggregates of interval data [14], [15], sliding window aggregates of stream data [16], [17], and efficient window aggregate computation by reducing I/O cost [18]. Big differences exist in the target data model between these works and mine, therefore the techniques exploited are quite different. Moreover, some graphic processing studies are related to our work, including special shaped window aggregates [19], [20], and convolution filter processing [21]. Still, these graphic studies are limited to 2-dimensional cases and only targeting average aggregate as 'window smoothing', while this work deals with data with any number of dimensions and compute all fundamental aggregates. Also data processing over complex shapes is not supported by all conventional array databases and lack applications in scientific fields. Therefore, they are beyond the scope of this paper.

The remainder of this thesis is organized as follows. Section 2 describes the background. Section 3 presents the proposal: recursive incremental computation method. Section 4 desribes design and implememtation. Section 5 explains the time and space complexity. Section 6 describes the evaluation. Section 7 discusses related work, and conclusions are provided in Section 8.

2. Background

2.1 Multi-dimensional Scientific Data

In many scientific fields, huge datasets with multiple dimensions are generated rapidly, pushing urgent requirements on efficient systems that can storage, manage and analyze such data. This paper proposes a efficient method to compute a particular multi-dimensional data processing task, which has real applications in many scientific fields.

MODIS is short for the Moderate Resolution Imaging Spectroradiometer [22], which is a data product from NASA's Earth Observing System program [23]. The MODIS data is gathered by two NASA satellites: Terra and Aqua [24], [25]. MODIS collects remote sensing data for earth science researches. The sensing data includes 36 spectral bands and 3 spatial resolutions, while being collected at rates up to 11 Mbps. Aqua and Terra both have polar orbits that enable them to encircle the Earth approximately every 90 minutes and view nearly the entire surface of the planet every 1-2 days.

MODIS data has three dimensions, longitude, latitude and time. It is provided to the scientific community free of charge, which makes it a good choice to run evaluation of our experiment on MODIS data. Furthermore, there are researches attempt to use scientific databases to handle such data, such as MODBASE [9]. In MODBASE work, an earth science benchmark including several analysis queries has been designed, among which our target operator improved in this work is also involved. In the evaluation, we execute experiments following this benchmark and check the performance improvement of the proposed method.

2.2 Array Database - SciDB

To efficiently store and analyze big multi-dimensional data, array database systems have been adopted in a variety of scientific applications, including those in cosmology, geo-informatics, and experimental physics. Array databases implement an array model as their basic data model to overcome the impedance mismatch problem incurred by the relational model.

Among current developed array database systems, SciDB is one of the most advanced. SciDB is open source and has been actively developed [2], [3]. It provides a parallel query processing feature on a cluster system for high performance. Array data is divided into small subsets, referred to as chunks, to deal with a large size of data that does not fit in physical memory. During query processing, only the necessary chunks are accessed from storage, which effectively avoids memory overflow.

2.3 Window Aggregate Operator

The window aggregate query over multi-dimensional data is a popular operator that is often used in the many scientific fields, such as meteorology [12] and earth science [9], [10]. A window aggregate operator calculates aggregate functions over a sliding window. It takes arguments as: the source array; the window sizes in each dimension that determine the window scope; the aggregate function over a particular attribute to be compute. The result of a window aggregate operator is an array with the same size and dimensions as the source array, while each output cell contains the aggregate result calculated over the window around the corresponding cell in the source array. An example of a window aggregate query is shown in Query 1 and Figure 1. The Query 1 here follows the synopsis of SciDB. Here, *arr* is the name of the input array, which have 2 dimensions and an attribute named 'v'. The aggregate function to compute is maximum.

Query 1 : window(arr, 0, 1, 0, 2, max(v))

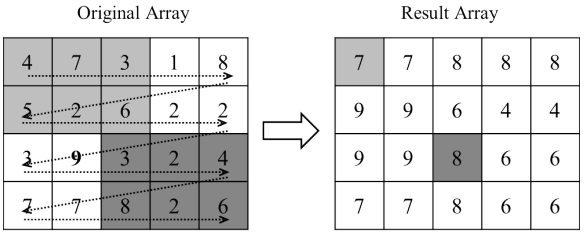

Fig. 1: Window Aggregate Example

In figure 1, the source array is shown on the left; the array on the right is the result. The dotted lines in the source array illustrate how the window moves. Each cell in the result array contains the maximum result of its corresponding window. Two cells in the result array and their corresponding windows are respectively marked with different shades of grey.

To process a window aggregate over multi-dimensional array data, SciDB handles each window independently. It scans each window, accumulates the values of all cells inside, and calculates the aggregate result. We refer to this method as a **naive method** because it is simple and straight forward.

This naive method involves a certain weakness in performance: during computation, redundant steps exist that waste computational resources. If we observe the windows in a window aggregate, it is easy to find that two neighbouring windows share a

large portion of same area. In order to eliminate such redundant calculation and improve the performance of window aggregate tasks, we propose incremental computation method in this work.

3. Proposed Method

In this section, details of the proposed method are introduced. First, the most simple case, the incremental computation of 1-dimensional window aggregate is explained along with detailed buffer tools designed for different aggregate functions. This 1-D case is the fundamental of our proposal. Then an straight forward attempt of applying the 1-D case to n-D case is simply explained, which has been evaluated and published in our previous work [26]. However, this solution has certain defects and is not good enough for n-dimensional cases. Finally, the proposal, recursive incremental computation method is explained, which further optimizes the calculation of n-D window aggregates. In the rest of this paper, item '**IC**' represents "incremental computation" for short.

3.1 Incremental Computation on 1-D window aggregate

Incremental computation of 1-dimensional window aggregate is the fundamental of this work. In order to reduce redundant calculation, intermediate informations of calculated windows are stored and reused when computing other windows with the help of certain tools. In this work, we refer to such tools as "buffer tools".

Buffer tools are responsible for maintaining and reusing intermediate results during the incremental computation process. A buffer tool should has a data structure as the "buffer" to maintain informations and provides interfaces that can update the buffer and most importantly be able to fetch demanding aggregate result using the information stored in the buffer.

The design of buffer tools is very important in this work. It directly affects how good improvement incremental computation can achieve comparing with naive method. Therefore in this subsection, detailed data structure design and how each type of buffer tool works in 1-dimensional array scenario is introduced first.

3.1.1 Buffer Tool Design Requirements

Despite different aggregate functions served and different data structures exploit as buffer, all types of buffer tools share the same pattern in their regular behaviours. In other words, the design of a buffer tool should meet several requirements so that it can help achieve incremental computation in high performance.

Figure 2 shows how a buffer tool helps to achieve incremental computation in 1-D window aggregate generally. It is obvious that two requirements are necessary to design an effective buffer tool.

The first one is efficient updating operators. As shown in Figure 2, when moving to a new window, the oldest cell a is no longer in current window, and one new cell b comes into current window. Of course, operators of insertion and deletion should be supported in a buffer tool so that it can update buffered data and maintain the correct values exactly representing the current window.

The second one is fast aggregate result fetching based on buffered data. A window aggregate operator is to compute aggregate value of every window. Therefore a buffer tool should support an operator of result fetch that returns the desired aggregate value of the current window after updating(insertion, deletion) is finished. This "result fetch" operator better works in constant time so that no extra calculation is required.

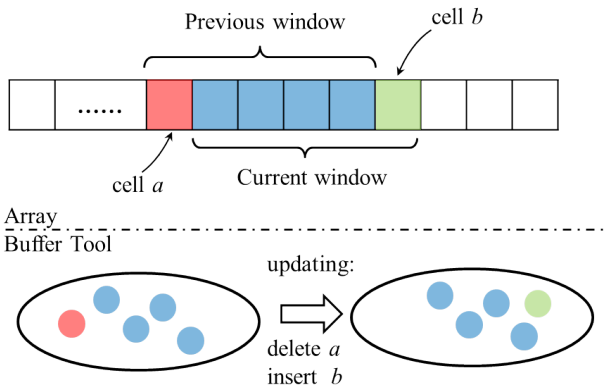

Fig. 2: Incremental Computation of 1-D Window Aggregate with Buffer Tool

To satisfy these two requirements, different data structures are designed to achieve incremental computation for different aggregate functions. In this paper, we improve six fundamental aggregate functions widely used in data analyzing tasks, including summation, average, minimum, maximum, variance and standard deviation. In the rest of this paper, these aggregates are also expressed in short as sum, avg, min, max, var and stdev. They are divided into 3 groups based on similarity in computation. Details of the buffer tools designed for each group are introduced one by one specifically in the rest of this subsection.

To be noticed that in this subsection, all the process are discussed under 1-D window aggregates case.

3.1.2 Summation & Average

Summation and average aggregate functions share almost the same computing process. To compute average aggregate, simply dividing the summation with number of cells in current window and the average result is calculated. Thus this group is discussed with sum as the represent.

A circle list-like structure is used to buffer data in the current window, referred to as "sum list" and an extra **SUM** temporary value is maintained to be the summation value of all elements in the list at any moment. This value serves to execute result fetch efficiently in $O(1)$ time as it can be directly returned.

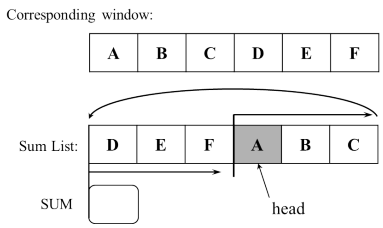

Fig. 3: Sum List Structure

As shown in the Figure 3 above, the buffer contains all cells' values of the current window, therefore the buffer size is same as the window size.

Let's consider how to compute a new window with sum list

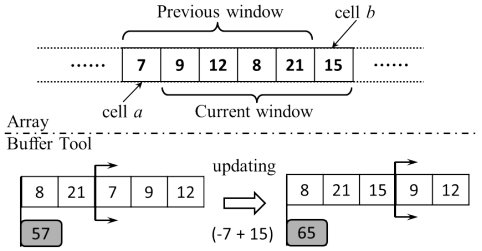

Fig. 4: Updating Details of Sum Aggregate

maintained. As shown in Figure 4, the oldest cell a is no longer in current window, it is removed from the buffer and subtracted from the SUM value; meanwhile a new cell b comes into the window and it is inserted into the buffer(replace cell a) and it is added to the SUM value. After these two updating operators, the SUM value is exactly the aggregate result which can be achieved immediately. As window moves on, the buffer list always contain exactly all the cells in the current window and the SUM value is always corresponding to the summation result. The arrows in the figure show the head of the sum list.

3.1.3 Variance & Standard Deviation

Variance and standard deviation are important evaluations in statistical analysis. They are supported in SciDB as aggregate functions and the proposed method also works on them as well. Standard deviation is the square root of variance, so we only discuss variance as the represent for this group.

Buffer tool is designed so that current window's aggregate result can be computed fast from the intermediate information buffered. For variance aggregate, it is more difficult compared with summation. Summation is a simple computation by adding up all the elements. Therefore when update occurs, the SUM value can be easily maintained by subtracting the deleted element, or adding the inserted element, either costs only $O(1)$.

When it comes to variance, a similar VAR value will not work because the computation of variance is more complicated. Variance of a sample dataset X is computed as follows, with μ standing for the average value of the dataset, as $\mu = \frac{1}{n}\sum x_i$.

$$Var(X) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2 \tag{1}$$

To find a way of computing variance result efficiently, let's deform the formula:(V is to represent variance value)

$$\begin{aligned} V &= \frac{1}{n-1}\sum(x_i - \mu)^2 \\ &= \frac{1}{n-1}\sum(x_i^2 - 2x_i\mu + \mu^2) \\ &= \frac{1}{n-1}(\sum x_i^2 - n\mu^2) \end{aligned} \tag{2}$$

Using S to represent $\sum x_i$ and SS to represet $\sum x_i^2$, we can get:

$$V = \frac{1}{n-1}(SS - \frac{1}{n}S^2) \tag{3}$$

From this equation, we can compute the variance in constant time during the window aggregate. The buffer tool is similar to sum list, with same buffer structure of circle-list maintaining all of current window's cells. Besides, two values S and SS are maintained. We call this buffer tool as "var list".

When it comes to a new window, the buffer is updated in the same way as sum list introduced in subsection 3.1.2. Along with the updating, value S and S^* can also be renewed in constant time. Considering insertion first, using v to represent the value of the new coming cell, and S^* and SS^* to represent the new values after updating. It's obvious that

$$\begin{aligned} S^* &= S + v \\ SS^* &= SS + v^2 \end{aligned} \tag{4}$$

Deletion is the same. After these two values are renewed, the var result of current window can be directly computed according to equation (3) in constant time. In this way, var and stdev can be incrementally computed. Because var list is almost the same as sum list except one more temporary value maintained, details of the structures are omitted here.

3.1.4 Minimum & Maximum

Minimum and Maximum are obviously similar and share the same buffer tool design. In the following discussion, we take "min" aggregate as the represent of this group.

One problem is that if we simply buffer all current window's cell values, a scan of the buffer is still necessary to get the minimum result, which is slow and drags the performance of incremental computation down to the naive method level. A temporary value similar to the SUM value can not work here because when removing an old cell, if it is the most smallest one, there is no way to find the minimum one among the remaining cells to update the min value in constant time.

We design a un-decreasing circle queue as the buffer tool to achieve efficient IC for min aggregate. We name it "min queue". The structure is shown in Figure 5. Similar to sum list, it has a head pointer and tail pointer for circling the queue to save memory consumed. In the queue, every element's value is smaller than previous elements, and each element is stored along with its position in the original input array. This extra position record is used in deletion for checking whether the element's corresponding cell is still in the current window or not.

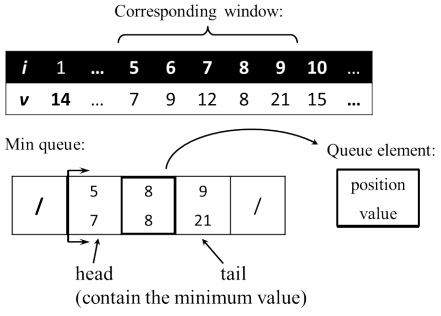

Fig. 5: Min Queue Structure

When moving to a new window, updates need to be done. To delete the oldest cell, it is required to check the queue's head element's position. If the head's cell is no longer in current window, remove it by simply moving on the head pointer one step. Then to insert the new cell, keep comparing the new value with the tail

element in the queue. If the tail is larger, remove it by subtracting the tail pointer and again repeat this process by comparing the new tail element. When it stops, the tail element is deem to be smaller than the new cell's value, and we insert the new cell into the buffer queue. By doing so, the elements inside the buffer are assured to be increasing. For the result fetch operator, the head element of the queue is the smallest one in the buffer, thus it is the minimum result of current window as well.

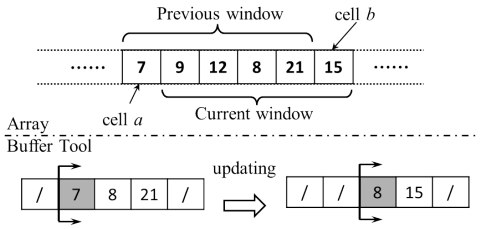

Fig. 6: Updating Details of Min Aggregate

This design works because an element already in the queue implies that it appears in earlier position than the new cell b, if it is not smaller than b, it has no chance to be the minimum value for any window after. Every following window certainly includes b as well. Therefore, there is no meaning to remain a cell whose value is larger then the tail in the buffer. Figure 6 shows a detailed example of min window aggregate about how it works incrementally with the assistance of the buffer tool, "min queue".

In the example, after moving on to a new window, the head element '7' is no longer in current window and it is deleted. Meanwhile, new cell's value '15' is to be inserted. At this point, the tail element in the queue '21' is bigger than 15 and it is obvious that any later window would at least have min value as 15, so it is meaningless to remain 21 in the queue. Therefore it is deleted. Then the new tail '8' is smaller than 15, so it can still be the min value as long as it is still in the window. The checking stops and 15 is inserted. After updating, the head element 8 is the min aggregate result of current window.

Maximum function is almost the same as minimum, except buffer tool is un-increasing queue instead of un-decreasing and opposite comparing operator is used during the process.

3.2 Incremental Computation Attempt on *n*-D scenario

As discussed in Subsection 3.1, the incremental computation of 1-D window aggregate has been solved. Because the scientific data we deal with has multiple dimensions, an efficient IC solution for n-D window aggregate is required.

With 1-D solution of window aggregate on hand, a natural idea is to divide the n-D task into multiple 1-D tasks. We can select one dimension to be the IC dimension, which is corresponding to the single dimension in 1-D case. Meanwhile, all the other $n-1$ dimensions are used to generate basic windows. Then for each basic window, window moves along with the IC dimension and the aggregate results of each window are calculated incrementally, just like the process of 1-D case. This 1-D subtask is the computation round of the corresponding basic window. This n-D solution has already been implemented and evaluated in our previous work [26].

3.2.1 Processing Details

From the description above, there are 3 important concepts need to be explained first so that the n-D scenario can be described in a clear and easy-understanding way. They are **basic window**, **computation round** and **window unit**.

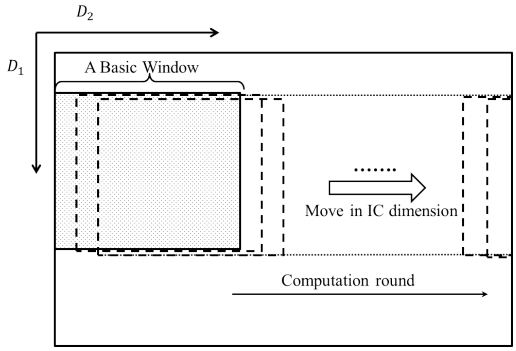

Fig. 7: Basic Window and Computation Round

As shown in Figure 7, a basic window is the start point of its corresponding computation round. During this computation round, window slides along with the IC dimension until all windows derived from this basic window are computed. In another description, an n-D dimensional source array has $\prod_{i=1}^{n} D_i$ windows to be computed. All these windows are divided into $\prod_{i=1}^{n-1} D_i$ groups. Each group of windows can be computed incrementally as a 1-D subtask. And the basic window is the first window in each 1-D incremental computation subtask. Here D_i stands for the size of dimension i.

In Figure 7, dimension 2 is the IC dimension and dimension 1 is used to generate basic windows. Thus there are at total D_1 basic windows' computation round to be processed. Each computation round is a 1-D subtask and the windows' aggregate results are computed incrementally with the help of buffer tools.

Recall the updating situation of 1-D case shown in Figure 2, when moving to a new window, only two cells are required to be updated (one to delete and one to insert). When it comes to a computation round in n-dimension case, multiple cells are required to be updated. Still these cells have a certain positional pattern and we use item "window unit" to describe such group of cells to be inserted/deleted.

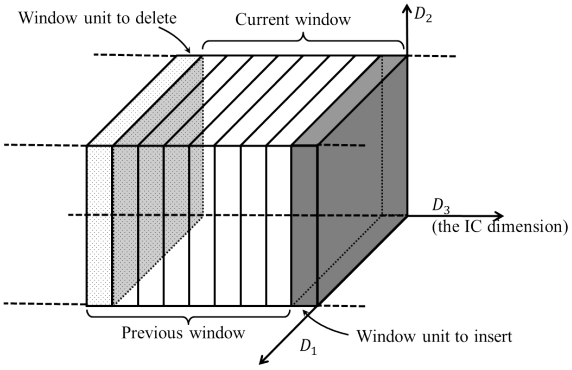

Fig. 8: Window Units to be Updated During a Computation Round

A window unit is a subset of window. It is named so because

window units are the smallest computing units in the process of multi-dimensional window aggregate incremental computation, just like cells are the smallest computing units in 1-D case. An n-D window is divided into multiple window units along the IC dimension. Each window unit is a slice of the original window.

Figure 8 shows the window units to be updated during a computation round. It is clear that window units have same behaviours as the single cells during the 1-D incremental computation process. Therefore the 1-D IC technique using buffer tools introduced before can be applied to the multi-dimensional case as well. Each basic window's computation round can be treated as one single 1-D window aggregate process. The only difference is that the values to update in buffer tools are the aggregate values of window units instead of values of single cells.

In this way, the n-dimensional window aggregate is executed along the IC dimension incrementally. It successfully reduce the redundant calculation exist in the IC dimension.

3.2.2 Defect: Redundant Calculation Still Exists

The weakness of this solution is that it only adopts the IC scheme in one dimension despite the fact that redundant work exists in all dimensions.

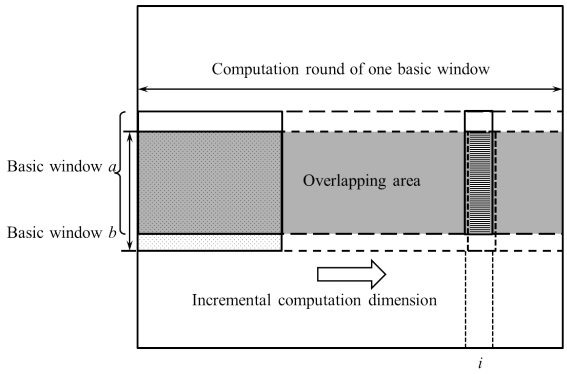

Fig. 9: Remaining Redundant Calculation in Basic IC

When computing the aggregate value of the new window unit, every cell inside it needs to be scanned. Considering two neighbouring basic windows and their computation rounds. Overlapping areas are processed multiple times as shown in Figure 9, with a 2-D example. In this figure, basic window a and basic window b are two adjacent basic windows. It is obvious that a certain area to compute in computation round of b has been already processed in the computation round of basic window a. To address this problem more clear, let us focus on position i in the IC dimension, and compare the new window units in these two computation rounds. Apparently, these two window units only have 2 different cells and the remaining cells are all the same.

3.3 Proposal: Recursive Incremental Computation Method

Recursive IC method is the proposal of this work. It manages to achieve incremental computation in all dimensions and completely eliminates redundant calculation. This is a solution designed targeting the multi-dimensional window aggregate.

3.3.1 Recursive Dimensionality Reduction

In order to remove the repeated work exist in scanning the window units, our solution is to execute the window aggregate **recursively** and somehow achieve IC in every dimension.

We design the recursive IC method to work in multiple levels. For a n-D window aggregate task, it is solved in n levels. The first level handles the n-D problem. One dimension is selected as the IC dimension in this level. In each moving step on this IC dimension, the calculation of new window units is equal to a window aggregate task with 1 less number of dimensions in next level. And in the last level, there are the 1-D window aggregate tasks. Because the results of window aggregate in next level is equivalent to the aggregate values of window units in current level, once the later level's IC process is finished, required window units' aggregate value in the current level can be immediately achieved and there is no need to scan them.

Therefore, the window aggregate can be done recursively. Every level has its unique incremental computation dimension and specifically focuses on the incremental improvement in that dimension. A n dimensional window aggregate's incremental computation is achieved with the help of multiple $n-1$ dimensional window aggregates. And these $n-1$ dimensional window aggregates are again incrementally solved exploiting the results of smaller $n-2$ dimensional window aggregates in next level. This process keeps going on until the dimension number is reduced to 1.

It should be noted that recursive IC method needs to maintain one buffer tool for each basic window. In a particular level, all basic windows' computation rounds must move together because their new window units are calculated together in the same computation round in next level. Therefore, in all levels, every basic window' round should maintain its own buffer tool.

3.3.2 Processing Details

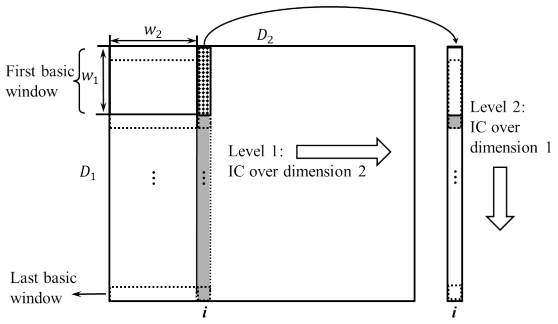

Fig. 10: Recursive IC of a 2-D Array

Figure 10 shows how recursive IC works with a 2-D window aggregate task. In the figure, D_i stands for the dimension size and w_i stands for the window size in each dimension. In the level 1, dimension 2 is selected as the IC dimension, thus there are D_1 basic windows in this level. All these basic windows' computation rounds are processing together, each one maintains one buffer tool. When it moves to position i in dimension 2, the new window units' aggregate values are required to update the buffer tools. The new window unit corresponds to the first basic window's round is marked with dotted background and every computation round has one new window unit to handle. Recursive IC method pushes this computation task of window units to level 2. In level 2, the task is a window aggregate on 1-D array which

is a slice from the original array and is mark in grey. This slice contains all the window units required in level 1 and they can be treated as 1-D windows in level 2. With dimension 1 as the IC dimension in level 2, they can be computed incrementally and results returned to level 1 to update the corresponding buffer tools.

For every moving step in level 1 on dimension 2, a complete incremental computation process in level 2 on dimension 1 is processed. In this way, level 1 and level 2 each accomplishes incremental computation on one dimension respectively. The whole process does not involve any redundant calculation at all.

Here is another 3-D example of how a multi-dimension window aggregate task is processed incrementally in all dimensions recursively.

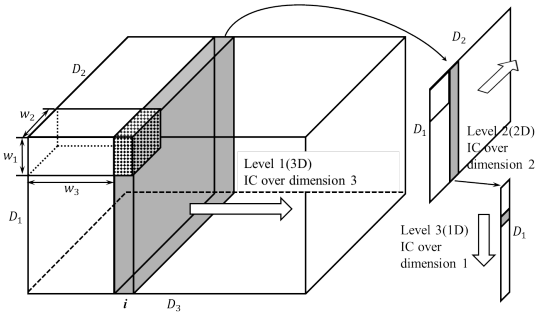

Fig. 11: Recursive IC of a 3-D Array

In level 1, a window has three dimensions and its corresponding window units can be treated as 2-dimensional windows. In each moving step along the IC dimension, new window units' aggregate values are required to update buffer tools and compute current windows' results. Again, a 2-D slice contains all these window units can be cut out and a 2-D window aggregate can exactly compute these window units. In the figure, one new window unit is marked with dotted background and the dimensionality reduced slice is marked in grey colour. 2-D window aggregate over this slice is processed in level 2. Again, as shown above, computing the new window units in this 2-D window aggregate can be further broken down into 1-D window aggregates and handled in level 3.

In brief, the 3-D window aggregate task is broken down into D_3 2-D window aggregate tasks. Each 2-D window aggregate task is further broken down into D_2 1-D window aggregate tasks. The recursive dimensionality reduction finishes at this point because incremental computation of 1-D window aggregate is perfectly solved in subsection 3.2, as window units of 1-D window are just single cells, whose values can be accessed directly as aggregate values.

For window aggregate with more number of dimensions, the process is similar. Any number of dimensions can be recursively broken down and eventually reduced to window aggregate tasks with one single dimension. To be noticed that the selection of IC dimension in each level has no specific order. In the example above, level 1 can also choose dimension 1 as IC dimension, level 2 moving on dimension 2 and level 3 moving on dimension 3. Any order works as long as each dimension is selected once and only once during the whole process.

Although recursive incremental computation method improves the performance greatly, it is obvious that it consumes much more memory. This is a tradeoff by consuming more space to obtain time efficiency. However, the extra space cost is actually acceptable since the SciDB's chunking storage mechanism.

4. Design and Implementation

This section introduces more design details of our proposed method when implementing it into SciDB. As the system to implement and evaluate the proposal, several issues of SciDB, such as its storage management and plugin mechanism are also introduced as they are involved to detailed design and implementation.

4.1 Plugin Mechanism of SciDB

To evaluate the performances of the recursive IC method against the naive method, we implement it on SciDB. A straightforward method to implement a new operator with high efficiency is to add a new built-in operator in SciDB system. This requires an immense effort for both the implementation and maintenance.

To avoid the overhead, we employ the plugin mechanism supported by SciDB. It supports implementation of a user-defined operator(UDO). Users can implement their own operators that realize any data processing tasks using the C++ language through the plugin mechanisms. It is determined that the implemented plugin operators are registered in the SciDB library system. Following this mechanism, once the plugin library is loaded into the system, the operator can work exactly the same way as the built-in operators of SciDB for data accessing, processing and operator-result fetching. Therefore, the usage of plugin does not sacrifice performance and provides excellent maintainability.

For implementing a UDO with the plugin mechanism, two types of program files are necessary. They can be denoted as "LogicalMyOperator.cpp" and "PhysicalMyOperator.cpp" . In the file name, "MyOperator" stands for the name of the new operator. The "Logical" file is a sort of metadata that merely describes the input and output array schemas of the new operator. The "Physical" file describes data processing details that generates the result of the operator. Moreover, file "plugin.h" provided by SciDB should be included when building the UDO to notify the system that the code is user-defined-operator.

4.2 Window Aggregate : Iterative Pipeline Operator

When implementing UDO plugin into SciDB, one rule from SciDB is that if an operator's output array is possible to be large, this operator should work as an iterative pipeline. Because arrays in SciDB are generally huge. With such huge array as input, the result array returned by UDO is often huge as well. Therefore, it is hard to maintain the result array all in memory during the computation. To solve this issue, SciDB demands that such UDO should work as iterative pipeline. To execute such UDO, SciDB iterate every cell in the result array. When iterating to a new cell, a method about how to compute this new cell's result should have been implemented. The whole process is like a pipeline and results are computed online and returned (output to the screen/ save to a file/ save to array in storage) cell by cell.

Window aggregate is exactly such an operator. The result array has the same size as the source array, which is reasonable

to be huge. Because this special requirement from SciDB, when implementing the recursive IC method into SciDB, the order of how the result cells in the output array are computed should be reorganized.

4.3 Details in Implementation

As introduced in background, SciDB divides huge arrays into small chunks and distributes them within a cluster. Operators can be efficiently executed chunk by chunk without extra disk I/O. When processing a chunk, window aggregate operator works as an iterative pipeline. Therefore, how to compute the result of current cell when iterating the chunks in the result array is the most important part of the implementation. The method is referred to as "calculateNextValue()". When iterating to a new cell in the result chunk, SciDB calculates the value for the cell using this method. We show the implementation in the form of pseudo-code for both the naive method and the proposed recursive IC method.

In all the pseudocode, *cellPos* contains the multi-dimensional coordinators of the current cell to compute in the result array. Items *winFirst* and *winLast* contains the window area parameters.

Algorithm 1 calculateNextValue() in **naive method**

```
1: for each i ∈ [1,numDim] do
2:     firstWinPos[i] ← cellPos[i] - winFirst[i]
3:     lastWinPos[i] ← cellPos[i] + winLast[i]
4: end for
5: aggregateor.clear()
6: for all cell c ∈ window(firstWinPos, lastWinPos) do
7:     aggregator.insert(inputChunk.access(c))
8: end for
9: nextValue ← aggregator.accumulate()
```

Pseudocode above describes SciDB's built-in implementation of the naive method. In the code, "aggregator" is a class already implemented by SciDB that computes the aggregate value of a set of data. It supports "insert", which is used to feed one new element into the set, and "accumulate", which returns the aggregate result of the current dataset, including all elements inserted so far. In addition, "inputChunk" is also a built-in class provided by SciDB. It manages the chunks that are processed by current executing operator. The chunk data required for data processing is automatically fetched into the memory. Therefore, users are not required to explicitly focus on storage access. It supports the "access()" method to obtain the data in a specific cell based on its multi-dimensional coordinates.

Each time the result array iterates to a new cell, the method "calculateNextValue()" shown in Algorithm 1 is invoked to compute the aggregate result of the window corresponding to the cell. The scope of the window corresponding to the current cell is computed first (line 1-4). Then all of the cells inside this window are scanned and inserted into a aggregator(line 5-8). They are finally accumulated into the aggregate result of the current window.

Algorithm 2 shows how recursive IC method works. In the code, "bufSet" stands for the set of all the buffer tools maintained in the recursive IC process. For a n-D task, there are n levels and in each level, every basic window have one buffer tool. Thus bufSet support "getBuf" method that returns the corresponding

Algorithm 2 calculateNextValue() in **recursive IC method**

```
1: for i = 1 → numDim do
2:     firstWinPos[i] ← cellPos[i] - winFirst[i]
3:     lastWinPos[i] ← cellPos[i] + winLast[i]
4: end for
5: startLvl ← numDim
6: while startLvl > 0 & cellPos[startLvl] = firstPos[startLvl] do
7:     startLvl ← startLvl - 1
8: end while
9: for i = startLvl+1 → numDim do
10:    recursivePrepare(i, firstWinPos[i], lastWinPos[i]-1)
11: end for
12: cellValue ← inputChunk.access(lastWinPos)
13: nextValue ← recursiveUpdate(cellValue)
```

Algorithm 3 recursiveUpdate(curValue) in **recursive IC method**

```
1: curLvl ←numDim
2: offset ←0
3: while curLvl > 0 do
4:     curBuf ←bufSet.getBuf(curLvl,offset)
5:     fullWin ←firstPos[curLvl] + winLast[curLvl]
6:     if lastWinPos[curLvl] > fullWin then
7:         curBuf.delete()
8:     end if
9:     curBuf.insert(curValue)
10:    curValue ←curBuf.resultFetch()
11:    offset ←offset to locate corresponding buffer to update in
   higher level
12:    curLvl ←curLvl - 1
13: end while
14: res = curValue
```

buffer tool according to two given arguments: the level number and the desired buffer tool's offset value in that level.

Also, in our implementation, the incremental computation dimension of level i is set to be dimension i. Level 1 is the highest level which has the most basic windows with number $\prod_{i=2}^{n} D_i$, while level n is the lowest level which has only one basic window and of course one single buffer tool as well. The higher level's incremental computation depends on the results of one lower level's results as lower level's windows are the desired window units in higher level. Therefore, when moving to a new cell, the multi-level buffer tools are updated from the lowest level to the highest level. In the lowest level n, the new window unit is the new cell itself. After the level n's buffer tool is updated, its resultFetch method returns the aggregate result of current window in level n, which is also the new window unit in level n-1. Thus this new value can be directly used to update corresponding buffer tool in level n-1. This process keeps going on until the highest level's corresponding buffer tool is updated. This buffer tool's corresponding basic window's computation round is exactly at the position of current window needs to be computed in the result array. Then the result of resultFetch can be output to the pipeline.

First of all, current cell's corresponding window scope is computed and stored into *firstWinPos* and *lastWinPos*. Because the window aggregate operator works as an iterative pipeline, when the result array iterates to a new cell, only the last one cell in the corresponding window needs to be processed, while all other cells have already been done and related information stored in buffer tools. Line 12-13 access the value of this last cell and invoke method "recursiveUpdate" to recursively update all the

buffer tools in every level involved with this new cell. The recursive update is illustrated in Algorithm 3.

5. Time & Space Analysis

In this section, time and space complexities are analyzed for both methods of window aggregate introduced in this paper. For the preparation to describe the analysis, some parameters in window aggregates need to be defined first so that the description is consistent and clear.

For a n-dimensional array, its dimension sizes are defined as $D_1, D_2, ..., D_n$. Meanwhile, for a window aggregate query over such array, the window sizes in each dimension are specified as $W_1, W_2, ..., W_n$. According to this definition, the total number of cells in this array is $\prod_{i=1}^{n} D_i$, while total number of cells within a normal complete window is $\prod_{i=1}^{n} W_i$.

5.1 Naive Method

The naive method is the method used by most array databases currently to compute window aggregate. This method is straight forward and the time complexity is easy to analyze. Every window is processed independently, therefore each one can be considered separately. Focusing on one single window, all cells insides it need to be scanned and accumulated into the aggregate result. For example, if the aggregate function is maximum, all cells' values are compared to the max result and update the max value if current cell's value is larger. Because there are $\prod_{i=1}^{n} W_i$ cells within a window, the scan and update process of one single window obviously costs $O(\prod_{i=1}^{n} W_i)$. Also, because the windows are one to one mapping with the cells in the source array, there are at total $\prod_{i=1}^{n} D_i$ windows to be executed. Multiply them together and the total time complexity of naive method is

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

About space cost, because the naive method does not buffer any extra information and only process on original data from the source array, it has no extra space cost.

5.2 Buffer Tool Analysis

Because buffer tool is the common module used in the recursive IC methods, the complexities of buffer tools designed for all three aggregate groups are discussed here before analyzing the proposed method.

A buffer tool supports 3 methods, which are insertion, deletion and resultFetch. To be noticed that one buffer tool is corresponding to one basic window's computation round. During the computation round, the buffer tool is updated with new window units and returns current window's result in that computation round. Thus all the analysis of buffer tool below targets on such scenario with given IC dimension and given basic window, to discuss how buffer tool performs during the computation round.

First let us consider the time complexity. According to the details of buffer tools introduced in Subsection 3.1, method resultFetch's time cost of three groups are all the same as $O(1)$. For the update methods, as insertion and deletion, group sum/avg cost $O(1)$ because in each update method, only simple value set operators are invoked, once in the buffer and once on the temporary SUM value. Group var/std also cost $O(1)$ as the only difference is that this group has more temporary values to maintain but still it only requires simple value set operations.

However, min/max is a bit complicated. The deletion method also costs $O(1)$ but in the insertion process, there are several cells to be removed from the tail and the number of such cells to remove is not fixed. To consider the whole computing round together is a better strategy. During the whole computation round, at most D cells are removed when insertions take place. Here D is the size of IC dimension. Meanwhile, at total D insertions are executed during the computation round. Therefore, one single insertion costs constant time $O(1)$ in amortized analysis.

Then considering the extra space complexity of one single buffer tool. For group sum/avg, circle list is used and in every window move step, one cell is removed and one cell is inserted, therefore the total size of the list remains the same in most times. The only special situation is for the first W_m moves because there are only insertions and for the last W_m moves as there are only deletions. So the space required is $O(W_m)$, here W_m is the window size in the incremental computation dimension.

For group min/max, in every move step, if the first cell is no longer in current window, it is removed. Therefore, at any moment, the queue always only contains cells within the current window. So the number of elements in the queue during the process is at most W_m. Of course space cost is $O(W_m)$ as well.

For group std/var, similar circle list as sum/avg is used, thus buffer size is still W_m. Only difference is that more temporary values are maintained beside sum value, such as the count and the sum square value. However, the dominant cost still lays in the buffer and space cost is still $O(W_m)$.

Table 1: Complexity Analysis of Buffer Tools

Aggregate Function		sum/avg	var/std	min/max
Time	Insert	$O(1)$	$O(1)$	$O(1)^*$
	Delete	$O(1)$	$O(1)$	$O(1)$
	ResultFetch	$O(1)$	$O(1)$	$O(1)$
Space		$O(W_m)$	$O(W_m)$	$O(W_m)$

Table 1 summarizes the analysis result of all buffer tools involved in this paper. In the table, the insert of min/max group is shown with an superscript '*' meaning it is a result of amortized time analysis.

5.3 Recursive IC method

For a n-dimensional window aggregate, recursive incremental computation method executes in n levels. In the first level, with dimension 1 selected as the IC dimension, all the basic windows move on their own computation rounds. For each position in dimension 1, the windows require to get new coming window units' aggregate values so that each round's buffer tool can be updated and aggregate results of new windows can be calculated incrementally. To compute the window units' aggregate value, a n-1 dimensional window aggregate task in level 2 is invoked. Thus at total there are D_1 n-1 dimensional window aggregate tasks to process in level 2. Similarly, each task in level 2 invokes D_2 n-2 dimensional window aggregate tasks in level 3 and so on in a

recursive way.

Finally it comes to the 1-dimensional window aggregate in the lowest level with IC dimension as the nth dimension. The time complexity of 1-D IC process is $O(D_n)$. Then the aggregate results of windows in this task help compute the higher level's window aggregate invoked it as these windows are the window units required in that 2-dimensional task. Again this is a recursive updating process from level n to level 1, costing $O(1)$ in every level with updating and result fetching methods of the buffer tools. For each window in level n, it recursively updates one corresponding buffer tool in every level, thus the time cost for this recursive update process is $O(n)$.

From the analysis above, there are at total $\prod_{i=1}^{n-1} D_i$ 1-D window aggregate tasks in the level n, each such task has D_n windows whose results need to be updated recursively through all the n levels. Therefore the time complexity for recursive incremental computation method is

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

Then let us consider the extra space cost in recursive incremental computation method for the buffer tools. In level 1, there are $\prod_{i=2}^{n} D_i$ basic windows and during each basic window's computation round, one buffer tool with size $O(W_1)$ is used to maintain and reuse intermediate data. Then in level 2, there are $\prod_{i=3}^{n} D_i$ basic windows and each maintains one buffer tool with size $O(W_2)$. And in the last dimension, there is one basic window which maintains one buffer tool with size $O(W_n)$. Therefore, the total space cost is computed as follows.

$$W_1\prod_{i=2}^{n} D_i + ... + W_n < \prod_{i=1}^{n} D_i + ... + D_1 = O(\prod_{i=1}^{n} D_i) \quad (5)$$

5.4 Overall Comparison

Table 2: Time & Space Complexity Analysis Summary

	Time Complexity	Space Complexity
Naive method	$O(\prod_{i=1}^{n} D_i \prod_{i=1}^{n} W_i)$	none
Basic IC method	$O(\prod_{i=1}^{n} D_i \prod_{i=1}^{n-1} W_i)$	$O(W_n)$
Recursive IC method	$O(n\prod_{i=1}^{n} D_i)$	$O(\prod_{i=1}^{n} D_i)$

We summarize of the time and space complexity analysis in Table 2. "Basic IC method" refers to a simple incremental method we proposed in previous work[26]. From the analysis, it is obviously that in theory, the recursive incremental computation method is the fastest and the naive method is the most time-consuming one. While considering the space cost aspect, the basic incremental computation costs a bit more space than the naive method, and recursive incremental computation method costs considerable large extra space for the buffer tools. It is a tradeoff from space to time. The more information buffered, the more redundant calculation is avoided, and faster the method performs.

On the other hand, the extra space cost of recursive IC method seems to be a too heavy cost. However, with SciDB's chunking storage mechanism, this cost is acceptable. SciDB divides a huge array into small chunks that can be treated as sub-array of the original array with sizes can be fit in the memory. Therefore, when executing a query, it is processed in unit of chunk instead of the whole array data. Every time a chunk is loaded into the main-memory and the required data-processing tasks on this part of data are executed. In this way SciDB reduce the I/O cost. Window aggregate can also benefit from this mechanism. Every time only a chunk's window aggregate is to be calculated, therefore the extra space is also based on the chunk size instead of the whole array size. As one chunk can fit in main-memory, the space required for the buffer tools in recursive IC method also can easily fit in main-memory.

6. Evaluation

We evaluate the proposed recursive IC method against the naive method in SciDB. Another method, "basic IC method" [26] we proposed previously is also evaluated to show how recursive IC method further improve the efficiency. Two series of experiments are conducted, testing over real earth science data and synthetic data respectively.

6.1 Experiment Cluster Specifications

We build a SciDB cluster consist of 4 nodes to perform all the experiments in this work. Table 3 shows the cluster specifications and all the servers share the same machine configurations.

Table 3: Experiment Configurations

SciDB Version	14.12
Operating System	CentOS 6.5
Processors	Intel(R)Xeon(R)E56202.4GHz
RAM	24GB

6.2 Earth Science Benchmark

First series of experiment follows an earth science benchmark with real scientific application. A related work MODBASE [9], attempts to exploit SciDB to manage the earth science data and process analyzing tasks. In the earth science benchmark [27] proposed in MODBASE work, window aggregate operator is frequently used, especially executed over the result arrays of other analyzing tasks in order to produce arrays with lower resolution on purpose of visualization, comparison and further analysis. For experiment, we upload MODIS data into SciDB, following the MODBASE benchmark workflow and execute "Gridding Data" task from the benchmark which is exactly a window aggregate query.

6.2.1 Data Preprocess and Upload

45 MODIS files are downloaded with area of interest to be the US west coast, with collecting time during the year 2012. All these settings follow the data used in the MODBASE benchmark. Each MODIS file is about 160MB size. These files are in HDF format and the sensing data is stored in the satellite scanning order. Therefore, extra preprocessing works are required to map

the sensing data into corresponding cells located by longitude-latitude coordinators before loading into SciDB. Because this preprocessing work is not strongly related to the topic of this paper, more details are not included here.

After preprocessing, CSV files are produced, which can be loaded into SciDB. After upload, the earth data is in schema of a 3-dimensional array. The values of longitude and latitude dimensions are scaled 1000 times, so that different sensing samples will not overlap into same cell in the array. Thus, corresponding to the global area, longitude dimension is in scope of [-180,000, 180,000] and latitude dimension in scope of [-90,000, 90,000].

6.2.2 NDVI Gridding Task Evaluation

In the earth science benchmark, gridding task is executed on the result of other analysis tasks in order to down-sample the array so that the analysis result can be visualized. The gridding task actually consists of 2 steps. The first step is a window aggregate query to compute the average value of every cell on earth with the window scope to be $0.05° \times 0.05°$. In the second step, down-sample is conducted by selecting cells with fixed skip length. Of course, the gridding task consumes most time on the window aggregate step because it is computing-intensive, while the second step is fast and can be ignored comparing with the first step.

In this experiment, we select normalized difference vegetation index(NDVI) to be the analyzing task to produce input array of gridding task. We execute NDVI calculation following the source code in MODBASE work [22] as NDVI task is also included in the MODBASE benchmark. Three sets of area parameters are designed, which are $10° \times 10°$, $20° \times 20°$, $30° \times 30°$, corresponding to the area size on the earth to be analyzed. The output result of a NDVI analysis is a 2-D array with the same size as the area size settings. After NDVI analysis task is finished, window aggregate operator is executed over the result array. The result array has two dimensions and the window aggregate query in gridding task is as follows:

```
window(ndvi, 25, 25, 25, 25, avg(ndvi))
```

In this window aggregate query, for each cell, its window scope is to expand in both dimensions and in both directions by 25 cells. Thus the total window size is 51×51 cells as the central cell also counts. This is the same window setting in the MODBASE benchmark introduced previously, the window scope is about $0.05° \times 0.05°$, corresponding to 50×50 window on data after loaded into SciDB with longitude and latitude scaled by 1000.

The experiment has three groups corresponding to different size of area on earth to be processed in the NDVI task. Window sizes are varies from 11×11 to 51×51 in order to check how each method behaves. To be noticed that only the size of 51×51 window is required in the real earth science application.

Table 4: Metrics of NDVI Arrays

Area Size	Array Size	Cells	Density	Data Size
$10° \times 10°$	10000×10000	28787550	28.79%	559MB
$20° \times 20°$	20000×20000	90526766	22.63%	1.78GB
$30° \times 30°$	30000×30000	240706765	26.75%	4.32GB

Table 4 illustrates the details of the three source arrays used in the experiment. Because not every cell is filled in MODIS real data after the data mapping, there are empty cells in the array. In the table, "Array Size" stands for the total array scope. Column "Cells" contains the real number of cells that is not empty. Column "Density" shows how dense the array is.

Figure 12 shows the experiment result of the $10° \times 10°$ area.

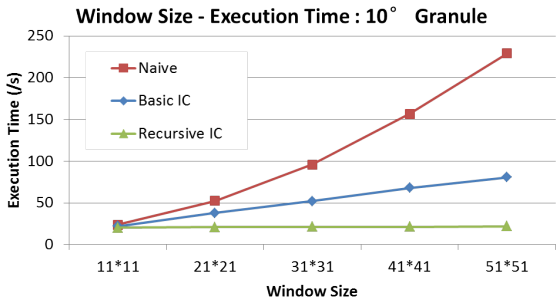

Fig. 12: Query Execution Time by Window Sizes, $10° \times 10°$ NDVI

The improvement is obvious. From the result, another interesting feature is that no matter how big the window is, the proposed recursive IC method's execution time is almost fixed. This is because in recursive IC method, all redundant calculation are completely eliminated, thus every cell in the array is processed only once. Therefore, the time consumed is not related to the size of window. On the contrary, Naive method travel every cells in all windows and its running time is increasing based on the total window size.

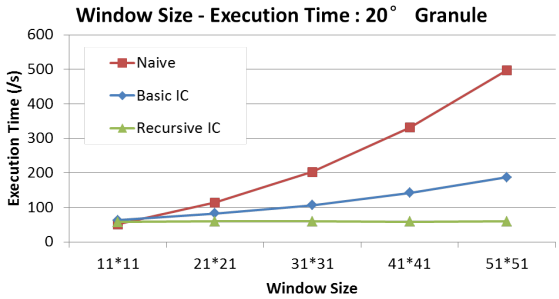

Fig. 13: Query Execution Time by Window Sizes, $20° \times 20°$ NDVI

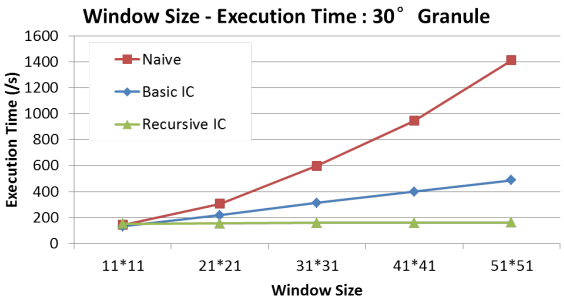

Fig. 14: Query Execution Time by Window Sizes, $30° \times 30°$ NDVI

Figure 13 and Figure 14 show the $20° \times 20°$ and $30° \times 30°$ cases. The results shows considerable acceleration has been achieved for the recursive IC method against the SciDB's built-in naive method. With the window size to be 51×51 as in the real application, recursive IC method achieves a speedup factor around 10

while basic IC method achieves a speedup factor about 3. In the largest $30° \times 30°$ case, the naive method needs more than 23 minutes and the recursive IC method only cost less then 3 minutes. Figure 15 below summarizes the experiment results only including the cases with window size settings in real application of the earth science benchmark as 51×51.

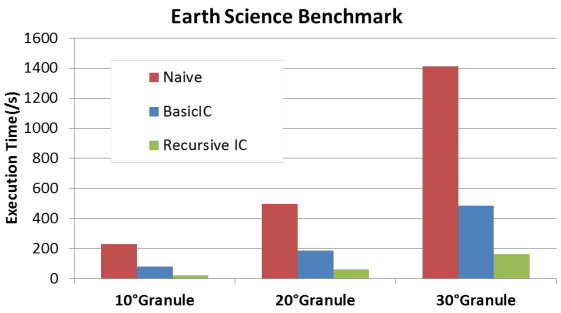

Fig. 15: MODIS Gridding Task Experiment Summary

Compared with time complexity analyzed in Section 5, the improvement seems not as big as in theory. This is because the result arrays of NDVI tasks are not very dense. The density can be found in Table 4. Our proposed method achieves best performance improvement when dealing with dense arrays. For sparse arrays, naive method can skip the empty cells but in incremental computation methods, during the IC computation round, every step is necessary as to update the buffer tools. Therefore, for a sparse array, the performance improvement will get some discount.

6.3 Synthetic Data Experiment

In order to thoroughly evaluate the performance of the methods, we generated several series of synthetic data to check how they perform with various settings of different parameters. The attribute values of arrays are randomly generated in range [0, 100000].

6.3.1 Parameter: Dimension Number

This series of evaluations are designed to check how the methods performs when array with more than 3 dimensions is processed. Because such high dimensional array is not included in the previous experiment of real earth science application.

Table 5 illustrates the array metrics designed in this evaluation. For all cases, array sizes are set to 1048576, and window sizes are set to 1024. Meanwhile, the number of dimensions increases from 2 to 5. Array sizes and window sizes are designed to be the fixed so that the overall workload is the same for the naive method in all cases. Besides, the schema of array and window are designed to be as evenly distributed as possible so that the importances of all dimensions are almost the same. This is to ensure the evaluation can expose the behaviours of methods when dealing with high-dimensional array. For example, a 2-D array whose schema is 10000×2 will mostly shown similar characteristic to a 1-D array with dimension size 10000, while a 2-D array with schema 100×100 behaviours differently for sure.

In the table, W_n stands for the window size in last dimension. This parameter is listed separately because it is the crucial factor makes performance different between basic IC method [26] and naive method.

Table 5: Evaluation Data Design to Check Dimension Number

Dimensions	Array Schema	Window Schema	W_n
2D	1024×1024	32×32	32
3D	$64 \times 64 \times 256$	$8 \times 8 \times 16$	16
4D	$32 \times 32 \times 32 \times 32$	$4 \times 4 \times 8 \times 8$	8
5D	$16 \times 16 \times 16 \times 16 \times 16$	$4 \times 4 \times 4 \times 4 \times 4$	4

All six aggregate functions are evaluated in this experiment. They are divided into 3 groups and aggregates in the same group has almost the same performance. Therefore, in each group only one function's execution time is displayed in the result as the represent of that group.

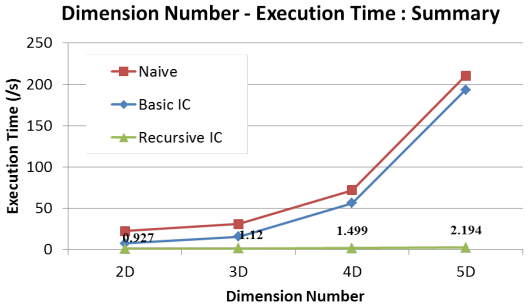

Fig. 16: Query Execution Time by Dimension Numbers, Summation Aggregate

Figure 16 shows the result of the sum/avg aggregate group. From the result, a very strange fact is that as the number of dimensions gets higher, the naive method surprisingly becomes very slow than expected. According to the data design, array size and window size are both fixed, thus the workload for naive method should remain the same in all evaluation cases. It means the execution time of naive method is supposed to be almost fixed. The reason caused this phenomenon is not clear. One assumption is that with more and more dimensions, the cost to locate and access one single cell is getting more and more expensive. In proposed recursive IC method, every cell is only visited once and this heavier access cost does not cause a big influence. However, in naive method, one single cell is visited so many times as all windows contain this cell has to access it once through the computing process. The basic IC method has the similar problem because the repeated cell accessing remains the same in most dimensions except the IC dimension. This makes the behaviours of naive method and basic IC method similarly slow when the number of dimensions gets big.

Here is results of the other two aggregate groups as shown in Figure 17 and Figure 18. The results show the same pattern. Actually, the execution time of different aggregate functions are so similar on same parameter settings that tiny differences can be shown in the figure when draw all of them together in a single figure. It's also consistent to the time complexity analyzed in chapter 5 that they all have the same complexity. Therefore, in the following experiments, we take variance as the represent of all these six aggregates and focus on the other interesting parameters.

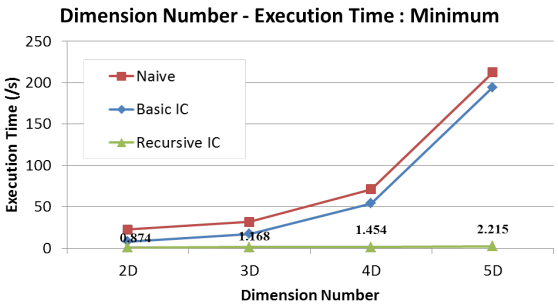

Fig. 17: Query Execution Time by Dimension Numbers, Minimum Aggregate

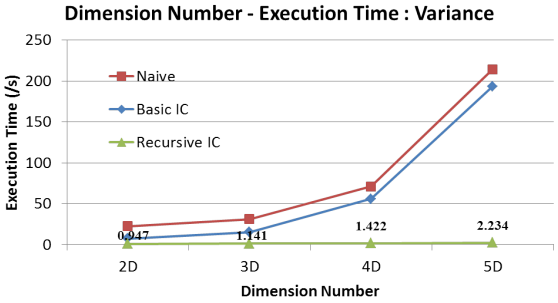

Fig. 18: Query Execution Time by Dimension Numbers, Variance Aggregate

6.3.2 Parameter: Array Size

This series of evaluation is designed to check how total sizes of the arrays affect the performance of window aggregate query. Six arrays with different size are generated. Table 6 illustrates the data design details of this evaluation.

Table 6: Evaluation Data Design to Check Array Size

Dimensions	Array Size	Array Schema	Data Size
2D	10^3	10×100	8.7KB
2D	10^4	100×100	87KB
2D	10^5	100×1000	869KB
2D	10^6	1000×1000	8.5MB
2D	10^7	1000×10000	86MB
2D	10^8	10000×10000	849MB

Three groups of evaluations with different window size settings are conducted, with window of 5×10, 10×10 and 10×50.

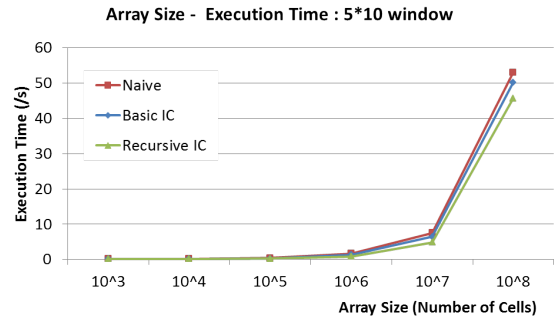

Fig. 19: Query Execution Time by Array Size: 5×10 Window

Figure 19 shows the result of 5×10 window case. Because the window size is quite small, the performance improvement is not so obvious. With larger window size cases shown in Figure 20 and Figure 21, the proposed method shows better performance.

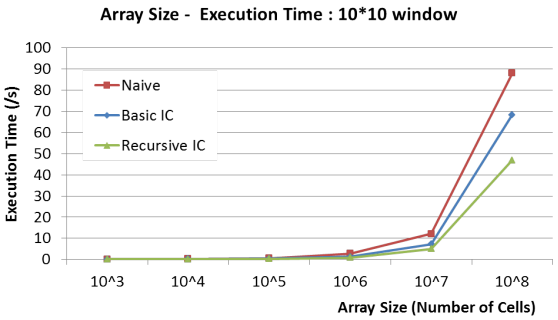

Fig. 20: Query Execution Time by Array Size: 10×10 Window

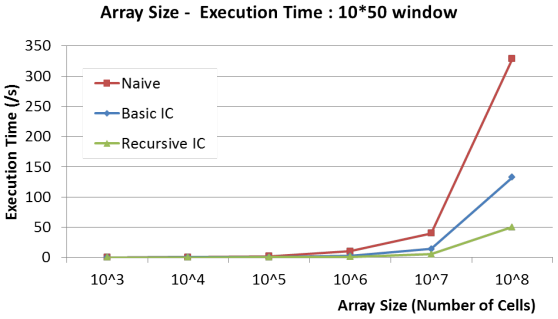

Fig. 21: Query Execution Time by Array Size: 10×50 Window

6.3.3 Parameter: Window Size

This evaluation is designed to check the parameter of window size. Because the amount of redundant computation is directly related to the window size, this parameter affects the improvement factor of our proposed method against the naive method greatly.

Three different sized 2D arrays are evaluated here, whose sizes are 1000×1000, 5000×5000 and 10000×10000. Because naive method consumes much time for large arrays, the window sizes designed for the later two array are relatively smaller.

The results are shown in the Figure 22, Figure 23, and Figure 24. From the result, it is obvious that with larger window, the proposed method achieve better improvement. Larger window means more redundant calculation exist in naive method, and our previous work, basic IC method can reduce such unnecessary calculation in one dimension. While our proposed recursive IC method in this work can completely eliminating such unnecessary calculation in all dimensions and greatly improve the performance. With the largest window 120×120 in Figure 24, recursive IC method achieves a speedup factor of 64.07, finishing a almost 2 minutes task of naive method in less than 2 seconds.

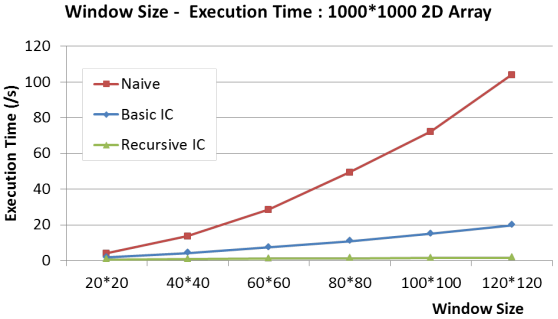

Fig. 22: Query Execution Time by Window Size: 1000×1000 Array

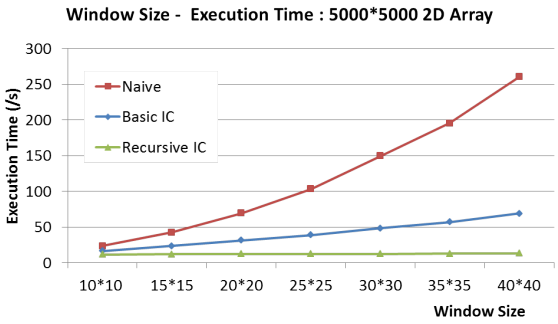

Fig. 23: Query Execution Time by Window Size: 5000×5000 Array

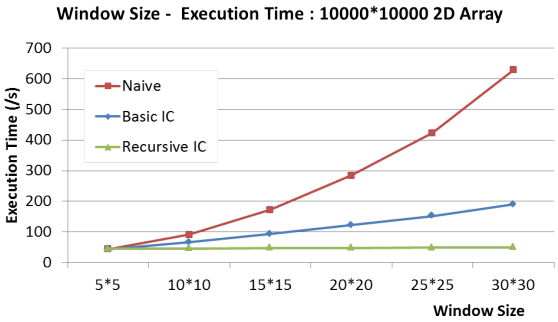

Fig. 24: Query Execution Time by Window Size: 10000×10000 Array

7. Related Work

Our previous works [26], [28] discuss a simple attempt to accelerate the execution of window aggregate by incremental computation. The proposed basic IC method in those works manages to achieve incremental computation in one dimension. However, the scientific data we are dealing with has multiple dimensions and the basic IC solution is not good enough as most redundant calculation still exists. One the other hand, the recursive IC method we proposed in this work further optimizes incremental computation of window aggregate, eliminates all redundant calculation in every dimension. It is a real n-dimensional solution and is a breakthrough compared with basic IC method whose improvement is limited in one single dimension.

Window aggregate is an important class of operators whose acceleration has been well studied. SAGA [18] presents efficient approaches for structural aggregates of array data. This work also deals with multi-dimensional data and the window aggregate. But it is focusing on reducing disk I/O cost . Our proposal exploiting incremental computation scheme to eliminate redundant calculation is out of the scope of SAGA. Besides, the improvement of SAGA is orthogonal to our work and thus can be adopted together. Their work is on the I/O level reducing unnecessary redundant data accessing exists in storage hierarchy, while our work is on the algorithm level reducing the redundant computation exists in query execution.

Incremental computation has been studied in the context of stream data processing [16], [17]. while performance improvement in these work is quite meaningful, they focus on 1D data and multi-dimensional problems are out of scope. On the other hand, stream data is quite different from array data which is prefetched and unchanged during query processing. Therefore, the "window aggregates" in stream data processing and array data processing are actually two different type of query.

For temporal aggregates of interval data, important works exist, such as balanced-tree [14] and SB-tree [15], to incrementally compute the query. The difference between these works and mine is the the underlying data model is different. The works introduced above are designed to deal with particular type of data, the interval data. They are not suitable for multi-dimensional array data, which is the target data type on which we herein focus.

A set of graphic processing studies are related to our work. The convolution filter exploits incremental computation and it implemented in OpenCV [21]. Nevertheless, it does not address large arrays, which do not fit into memory and it works only in 2-D cases. Some contest problems [29], [30] are more complicated than our problem, although they completely differ from the scenario of scientific data analysis tasks we try to solve in this work. Diamond, hexagon, and polygonal shaped window aggregates are discussed in [19], [20]; however, these algorithms are targeting 2-D images and they are not applied to n-D scientific data. Thus, such shapes are not supported by any array database systems, including SciDB. Furthermore, all the above works focus only on algorithms but lack a system design perspective and efficient solution for huge data that can't fit in memory. In this paper, on the other hand, we provide design and implementation of the proposed method on a distributed database system, SciDB.

About array databases, there are researches extending scientific features, such as data versioning [31] and uncertain data [32]. Also, efficient distributed storage and parallel processing of array data are discussed in work [33]. As the most popular array DBMS, SciDB gains more attentions [2], [3], [4]. These works focus on the architecture design and low-level array storage, while our work is to improve a specific query in array database.

8. Conclusion

This paper proposes an efficient algorithm for window aggregate operators in array databases based on an incremental computation scheme. Window aggregate is an important operator in array-oriented processing tasks and widely used in many scientific fields. However this operator is very computation-intensive and consumes long execution time. Based on our previous work, we propose a further improved method, the recursive incremental computation method to accelerate window aggregates by eliminating all the redundant works exist in every dimension.

We developed acceleration techniques for 6 aggregate functions that exploit different designs of buffer tools to efficiently eliminate redundant calculation: "circle-list" for summation, average, variance and standard deviation, while "monotone queue" for maximum and minimum.

Time and space complexity analysis is presented in the paper. To evaluate the proposed method, all methods are fully implemented into SciDB, which is a popular array database system. Experiments includes a real applications in earth science and synthetic data. The results of the experiments shows great improvement. Comparing with SciDB's built-in operator that implemented in naive method, the proposed method shows performance improvement by a factor of 10 for the MODIS earth sci-

ence benchmark. With a large window evaluation case over synthetic data, proposed recursive IC method shows a speedup factor of 64.

Acknowledgments This work is partially supported by JST CREST "System Software for Post Petascale Data Intenseive Science" and JST CREST "Extreme Big Data (EBD) Next Generation Big Data Infrastructure Technologies Towards Yottabyte/Year". It is also supported by JSPS KAKENHI Grant Number 25280043HA, JST CREST "Statistical Computational Comsmology with Big Astronomical Imaging Data" and JST A-STEP Grant Number AS262Z02895H.

References

[1] Stonebraker, M., Becla, J., DeWitt, D. J., Lim, K.-T., Maier, D., Ratzesberger, O. and Zdonik, S. B.: Requirements for Science Data Bases and SciDB., *CIDR*, Vol. 7, pp. 173–184 (2009).

[2] Cudré-Mauroux, P., Kimura, H., Lim, K.-T., Rogers, J., Simakov, R., Soroush, E., Velikhov, P., Wang, D. L., Balazinska, M., Becla, J. et al.: A demonstration of SciDB: a science-oriented DBMS, *Proceedings of the VLDB Endowment*, Vol. 2, No. 2, pp. 1534–1537 (2009).

[3] Soroush, E., Balazinska, M. and Wang, D.: Arraystore: a storage manager for complex parallel array processing, *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, ACM, pp. 253–264 (2011).

[4] Brown, P. G.: Overview of SciDB: large scale array storage, processing and analysis, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ACM, pp. 963–968 (2010).

[5] Zhang, Y., Kersten, M., Ivanova, M. and Nes, N.: SciQL: bridging the gap between science and relational DBMS, *Proceedings of the 15th Symposium on International Database Engineering & Applications*, ACM, pp. 124–133 (2011).

[6] Kersten, M., Zhang, Y., Ivanova, M. and Nes, N.: SciQL, a query language for science applications, *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, ACM, pp. 1–12 (2011).

[7] Baumann, P., Dehmel, A., Furtado, P., Ritsch, R. and Widmann, N.: The multidimensional database system RasDaMan, *ACM SIGMOD Record*, Vol. 27, No. 2, ACM, pp. 575–577 (1998).

[8] Vanderplas, J., Soroush, E., Krughoff, K. S., Balazinska, M. and Connolly, A.: Squeezing a Big Orange into Little Boxes: The AscotDB System for Parallel Processing of Data on a Sphere., *IEEE Data Eng. Bull.*, Vol. 36, No. 4, pp. 11–20 (2013).

[9] Planthaber Jr, G. L.: Modbase: A scidb-powered system for large-scale distributed storage and analysis of modis earth remote sensing data, PhD Thesis, Massachusetts Institute of Technology (2012).

[10] Stonebraker, M., Duggan, J., Battle, L. and Papaemmanouil, O.: SciDB DBMS research at MIT, *IEEE Data Eng. Bull.*, Vol. 36, No. 4, pp. 21–30 (2013).

[11] Brown, P. G.: A Survey of Scientific Applications using SciDB, *New England Database Day Program* (2015).

[12] Matsueda, M. and Nakazawa, T.: Early warning products for severe weather events derived from operational medium-range ensemble forecasts, *Meteorological Applications*, Vol. 22, No. 2, pp. 213–222 (2015).

[13] Jiang, L.: Source codes of the Proposed Incremental Computation Methods, `https://github.com/ljiangjl/Recursive-IC-Window`.

[14] Moon, B., López, I. F. V. and Immanuel, V.: Scalable algorithms for large temporal aggregation, *Data Engineering, 2000. Proceedings. 16th International Conference on*, IEEE, pp. 145–154 (2000).

[15] Yang, J. and Widom, J.: Incremental computation and maintenance of temporal aggregates, *The VLDB Journal*, Vol. 12, No. 3, pp. 262–283 (2003).

[16] Li, J., Maier, D., Tufte, K., Papadimos, V. and Tucker, P. A.: No pane, no gain: efficient evaluation of sliding-window aggregates over data streams, *ACM SIGMOD Record*, Vol. 34, No. 1, pp. 39–44 (2005).

[17] Wu, Y., Maier, D. and Tan, K.-L.: Grand challenge: SPRINT stream processing engine as a solution, *Proceedings of the 7th ACM international conference on Distributed event-based systems*, ACM, pp. 301–306 (2013).

[18] Wang, Y., Nandi, A. and Agrawal, G.: SAGA: array storage as a DB with support for structural aggregations, *Proceedings of the 26th international conference on scientific and statistical database management*, ACM, p. 9 (2014).

[19] Sun, C.: Diamond, hexagon, and general polygonal shaped window smoothing, *Proc. VIIth Digital Image Computing: Techniques and Applications, Sydney* (2003).

[20] Sun, C.: Moving average algorithms for diamond, hexagon, and general polygonal shaped window operations, *Pattern recognition letters*, Vol. 27, No. 6, pp. 556–566 (2006).

[21] García, G. B., Suarez, O. D., Aranda, J. L. E., Tercero, J. S., Gracia, I. S. and Enano, N. V.: Learning Image Processing with OpenCV (2015).

[22] NASA: Moderate Resolution Imaging Spectroradiometer, `http://modis.gsfc.nasa.gov/`.

[23] NASA: Earth Observing System Program, `http://eospso.gsfc.nasa.gov/`.

[24] NASA: Terra Earth-observing satellite mission, `http://terra.nasa.gov/`.

[25] NASA: Aqua Earth-observing satellite mission, `http://aqua.nasa.gov/`.

[26] Jiang, L., Kawashima, H. and Tatebe, O.: Incremental window aggregates over array database, *Big Data (Big Data), 2014 IEEE International Conference on*, IEEE, pp. 183–188 (2014).

[27] Stonebraker, M., Duggan, J., Battle, L. and Papaemmanouil, O.: Earth Science Benchmark over MODIS data, `http://people.csail.mit.edu/jennie/elasticity_benchmarks.html`.

[28] Jiang, L., Kawashima, H. and Tatebe, O.: Implementing Incremental Aggregate Computation on SciDB, *Research report of system software and operating system(OS)*, Vol. 2014, No. 2, pp. 1–8 (2014).

[29] UVAOnlineJudge: Maximum Sum Problem, `https://uva.onlinejudge.org/external/1/108.html`.

[30] ACM ICPC, J. D.: Square Carpets,, `http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1128` (2003).

[31] Seering, A., Cudre-Mauroux, P., Madden, S. and Stonebraker, M.: Efficient versioning for scientific array databases, *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, IEEE, pp. 1013–1024 (2012).

[32] Ge, T. and Zdonik, S.: Handling uncertain data in array database systems, *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, IEEE, pp. 1140–1149 (2008).

[33] van Ballegooij, A., Cornacchia, R., de Vries, A. P. and Kersten, M.: Distribution rules for array database queries, *Database and Expert Systems Applications*, Springer, pp. 55–64 (2005).