Electronic Preprint for Journal of Information Processing Vol.24 No.6

Regular Paper

Efficient Window Aggregate Method on Array Database System

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

Received: December 20, 2015, Accepted: July 5, 2016

Abstract: An array database is effective for managing a massive amount of sensor data, and the window aggregate is a popular operator. We propose an efficient window aggregate method over multi-dimensional array data based on incremental computation. We improve five types of aggregates by exploiting different data structures: list for summation and average, heap for maximum and minimum, and balanced binary search tree for percentile. We design and fully implement the proposed method in SciDB using the plugin mechanism. In addition, we evaluate the performance through experiments using the synthetic and JRA-55 meteorological datasets. The results of our experiments on SciDB are consistent with our analytic findings. The proposed method achieves a 17.9x, 12.5x, and 10.2x performance improvement for minimum, summation, and percentile operators, respectively, compared with SciDB built-in operators. These results align with our time-complexity analysis results.

Keywords: multi-dimensional array, incremental computation, array database, window aggregate

1. Introduction

Ubiquitous computing provides a dizzying array of data on our health movements, and changes in the environment [38]. The number of micro-sensing devices for the Internet of Things in daily life has been rapidly increasing [33]. The sizes of advanced sensing devices for scientific discovery, such as telescopes [32] and those used in experimental physics [31], are increasing. Sensing systems must continually collect and analyze these devices. These sensing data inherently include spatio-temporal attributes, which are naturally represented as multi-dimensional arrays. For example, meteorological data such as the JRA-55 [14] are represented as two-dimensional (2D) arrays with X and Y coordinates.

To manage such arrays, adoption of the relational 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 computation time for analytical tasks and in data management on account of impedance mismatching [5]. To efficiently store and analyze such multi-dimensional data, array database systems, such as SciDB [1], [2], [3], [4], [5], SciQL [6], [7], [8], and RasDaMan [9], 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 and experimental physics [19], [36].

An important class of queries in the array database is the window aggregate. It executes aggregate operators over a sliding window. Here, a window is more like a sub-array of the original array; its sizes in each dimension are specified by users. The window starts at the first grid of the array and moves in stride-major order from the lowest to highest value in each dimension. An example of a window aggregate query is shown in Query 1. The notation is based on SciDB [1], [2], [3], [4], [5]. This query computes the maximum aggregate with a sliding window over a 2D array. **Figure 1** depicts the way in which the query is processed in SciDB. The window size is set as 2×3. In the figure, the original array is shown on the left; the array on the right is the result. The dotted lines in the original array illustrate how the window moves. Each cell in the result array contains the result of the maximum computation for its corresponding window. The two result cells and their corresponding windows are respectively marked with different shades of grey. When the window size is increased, the execution time rapidly increases.

Query 1
SELECT max(v) FROM array1 GROUP BY window(2,3)

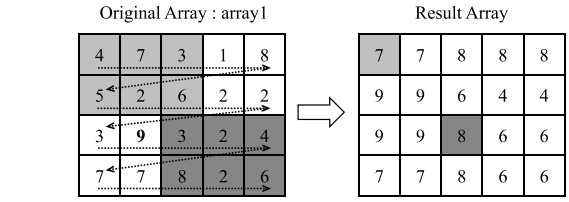


Fig. 1 Window aggregate by Query 1.

In this paper, we address the acceleration of window aggregates on array database systems. The contributions of this paper are twofold. First, we propose efficient algorithms for five types of window aggregates: maximum, minimum, summation, average,

[1] Graduate School of Systems and Information Engineering, University of Tsukuba, Japan on submission, Tsukuba, Ibaraki 305–8573, Japan
[2] Center for Computational Sciences, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
[†1] Presently with Google Dublin
[a)] kawasima@cs.tsukuba.ac.jp

and percentile. Second, we elucidate the scheme to effectively design and implement the proposed algorithms on SciDB, which is a typical array database system.

For the algorithm proposal, we exploit different data structures according to different operators to implement the scheme. For summation and average, we apply the list to maintain the intermediate results. For the minimum and maximum, the problem becomes more complex. We therefore apply the heap instead of the list to maintain the intermediate results. For the most complicated operator, the percentile, we apply the balanced binary search tree [15], [25]. In addition, for each window aggregate operator, we analyze the time complexities of our proposed method and the method implemented in SciDB.

For the second contribution, we exploit the plugin mechanism in SciDB for effective design and implementation of the proposed method. We show that the implementation of our method in SciDB demonstrates higher efficiency performance compared with the built-in operators in SciDB. All of our code described in this paper is available on GitHub [17], [18]; thus, all SciDB users can employ it.

Some database studies are related to our work. Examples include temporal aggregates of interval data [10], [11], sliding window aggregates of stream data [20], [22], and efficient window aggregate computation by reducing I/O cost [12]. The underlying data structures in these studies differ from those of our work. Moreover, some graphic processing studies relate to our work, including those using diamond, hexagon, and polygonal shaped window aggregates [26], [27], a convolution filter [28], and problems at programming contests [29], [30]. Such data processing over complex shapes is not yet supported by all conventional array databases, including SciDB. Therefore, they are beyond the scope of this paper. Finally, this paper is the extension of our report in a conference [21], and this paper has a substantial difference in experiments, design details, and related work from the previous work.

The remainder of this paper is organized as follows. Section 2 describes the background. Section 3 outlines our proposed methods. Section 4 presents our time complexity analysis. Section 5 details our design and implementation. Section 6 describes our evaluation. Section 7 discusses related work, and our conclusions are provided in Section 8.

2. Background

2.1 Array Database System

To efficiently store and analyze multi-dimensional sensor data, array database systems have been adopted in a variety of sensing systems, including those in cosmology, geo-informatics, and experimental physics [19], [36]. Array databases implement an array model as their basic data model. Array data can be expressed by a relational model; however, the expression is not intuitive. To overcome the impedance mismatch problem incurred by the relational model [5], array database systems, such as SciDB [1], [2], [3], [4], [5], SciQL [6], [7], [8], and RasDaMan [9], have been studied.

Among these systems, the most advanced array database system is SciDB [1], [2], [3], [4], [5]. SciDB is open source software and has been actively developed [36]. It provides all of the above operators as well as a parallel query processing feature on a cluster system for high performance. Array data are divided into a small portion of a subset, referred to as a chunk, to deal with a large size of data that do not fit in physical memory. During query processing, only the necessary chunks are read from storage, which effectively avoids memory overflow.

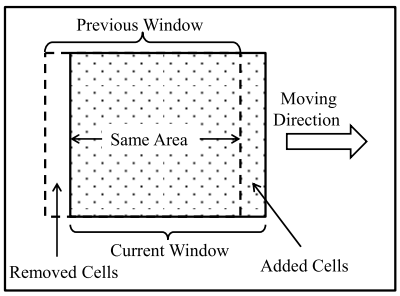


Fig. 2 Comparison of adjacent windows.

Similar to a relational database system, an array database system, such as SciDB, provides built-in operators and a query processor to enable users to intuitively issue SQL-like queries for processing arrays. An issued query is translated into a processing plan composed of operators. Then, through a processing plan, the result of the query is returned to the user. The difference between relational operators and array operators are the input and output. Relational ones deal with relational data, whereas array ones handle array data. Thus, the relational database is closed in relations, whereas the array database is closed in arrays and is deemed array complete.

2.2 Naive Window Aggregate Method

The window aggregate function over multi-dimensional data is a popular operation that is often used in the field of meteorology [16], [34], [37] and other fields. An example of a window aggregate is shown in Query 1 and Fig. 1.

To process a window aggregate over multi-dimensional array data, SciDB handles each window independently. SciDB scans each window, accumulates the values of all cells in each window, and calculates the aggregate result. We refer to this method as a 'naive method.' The naive method can involve a weakness in performance. During computation, redundant steps may exist that waste computational resources. If we observe the windows in a query, it is possible to find that two neighboring windows share a large portion of the same area. We show the scenario in **Fig. 2** with a 2D array case.

2.3 Challenge

The naive method ignores this feature of adjacent windows and separately computes each window. If one somehow reuses the processed data in previous windows to compute the aggregate in the current window, aggregate processing would be more efficient.

The approach to reusing intermediate data is not obvious for two reasons. First, a variety of operators exist because the window aggregation includes different types of specific operators: maximum, minimum, summation, average, and percentile.

Different algorithms and data structures for different operators should be applied; however, such algorithms and structures are not obvious. The second reason involves the implementation to SciDB. Although SciDB is the most advanced array database system, the method of implementing user-defined operators is not widely known. A straightforward method of implementing the new operator with high efficiency is to add a new built-in operator in SciDB. This would require am immense effort for both implementation and maintenance.

In this paper, we address the two above problems. We first develop efficient algorithms for window aggregate operators. Then, we present their effective implementation methods for SciDB.

3. Proposed Method

3.1 Incremental Computation Scheme

To exploit the feature of adjacent windows, we adopt an incremental computation scheme [23]. The central idea is to maintain the intermediate aggregate results of the current area and reuse them when computing the next area by eliminating redundant computations.

We improve the five types of the aggregate operators of the window aggregate queries. They are summation, average, minimum, maximum and percentile. For different operators, different data structures are used to efficiently maintain and reuse the intermediate results. Additional algorithm details about each operator are described in Section 3.

In general, the process of incremental computation for a window aggregate is divided into two major stages. The first stage is to move the basic window in the first $N-1$ dimensions. Here, a basic window is the starting point of the window movement in an incremental computation round, as shown in **Fig. 3**. After a new basic window is reached, the second stage begins. It moves the window along with the last dimension and incrementally computes the aggregate results for each window. After all windows derived from the basic window are processed, the given computation round is completed. Then, we must return to the first stage. We transition to a new basic window. The two stages are repeated until all the computation rounds of the basic windows are processed. At this point, the whole window aggregate query is completed.

We introduce step-by-step details on how the incremental computation methods perform window aggregating using five aggregate operators: summation, average, minimum, maximum, and percentile. For different aggregate operators, we select appropriate data structures to respectively maintain intermediate results. Accordingly, we can reuse them and efficiently compute the windows. For summation and average, we exploit the buffer list; for minimum and maximum, we exploit the heap; and for the percentile, we exploit the balanced binary search tree. To more explicitly yet simply describe the items in the remainder of the paper, we use 'SUM' for summation, 'AVG' for average, 'MIN' for minimum, 'MAX' for maximum, and 'PCTL' for percentile.

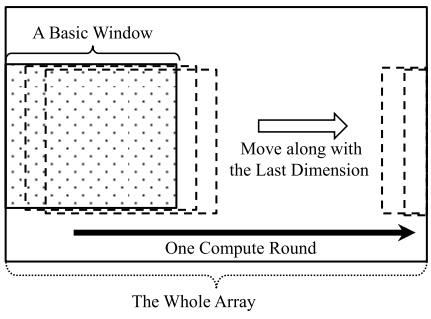


Fig. 3 Compute round in incremental computation.

3.2 Summation and Average

The calculation details of SUM and AVG aggregate operators are almost the same. This is because the AVG value of a window is computed by dividing the SUM by the size of the calculating window. Therefore, we together consider these two aggregate operators.

The main goal is to reuse the SUM values already computed in previous windows. Here, we propose adoption of a list structure to store intermediate results previously computed. We refer to this list to as 'SUM-list.' This list contains the SUM values of every window unit that belongs to the current window. When transitioning to the next window, only one new window unit must be computed; the other window units are already computed and can be directly obtained from SUM-list. Following is an example that demonstrates the operation of the proposed method with a small 2D array. The querying window size is set to 3×3.

Step 1: Generate a basic window and scan each window unit of it. Compute the SUM values of these window units and initialize SUM-list. **Figure 4** shows the details.

Step 2: Move the window along with the last dimension, which is dimension Y in **Fig. 5**. Step on one window unit; then, a new derivative window is generated. Scan the new window unit, calculate its summation, and update SUM-list by replacing the value of the oldest window unit. Figure 5 illustrates this process. After the update, the total SUM of the values in SUM-list is exactly the aggregate summation of the current window.

Step 3: Proceed forward, and calculate the aggregate value of all the windows derived from the basic window in Step 1.

Step 4: Continue to move the basic window down to obtain new basic windows. For each basic window, repeat Steps 1 to 3, which is a compute round. After finishing the compute rounds of

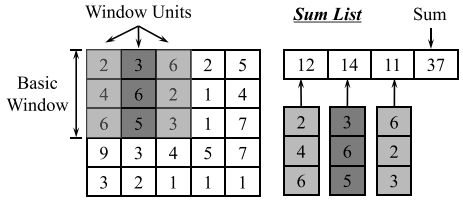


Fig. 4 Initialize basic window and SUM-list.

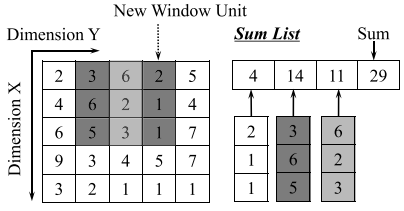


Fig. 5 Process a new unit and update SUM-list.

all the basic windows, the window aggregate query is completed.

For the arrays with more than two dimensions, the process of incremental computation is the same. It uses SUM-list to maintain SUMs of window units in the previously calculated window. Then, there is no need to compute most of the window units in the current window because aggregate SUM is directly obtained from SUM-list. Only the new window unit must be scanned.

3.3 Minimum and Maximum

For MAX and MIN, the incremental computation becomes more complex compared with SUM/AVG. If we adopt a buffer sequence such as SUM-list, which is used in SUM/AVG operators, we must still scan this buffer list to obtain the MIN/MAX value of the current window. The scan execution is costly when the queried window size is large.

To solve this problem, we propose application of the heap data structure instead of the list to maintain the MAX/MIN values of the current window. Adopting the heap saves time in calculating the MIN/MAX values. In the explanation below, we describe the MIN function as an example. The window size of the query is set as 3×4.

Step 1: Generate a basic window. Scan the window units in the basic window and compute the MIN value of each window unit. Insert them into a heap for MIN. For example, in **Fig. 6**, the MIN value of each window unit is marked in bold. They are 3, 6, 15, and 7, respectively.

Step 2: Move the window along with the last dimension (dimension Y). Step on one window unit and obtain a new window. Scan the new window unit and calculate its MIN value. Then, insert the value into the heap. **Figure 7** details this step.

Step 3: Check the heap root and remove it if its corresponding window unit is no longer present in the current window. Repeat the procedure until a window unit corresponding to the root node is within the current window. Then, the value of the root node is exactly the MIN value of the current window.

Step 4: Continue to proceed forward, and calculate the MIN of all the windows derived from the basic window in Step 1.

Step 5: Continue moving the basic window down to obtain new basic windows. For each basic window, repeat Steps 1 to 4, which is a compute round. After finishing all compute rounds of the basic windows, the whole window aggregate query is completed.

For the window aggregate of MIN/MAX, the core scheme of the incremental computation remains the same as SUM/AVG. However, the adoption of the heap data structure instead of the list speeds up the process of reusing the buffered intermediate aggregate results.

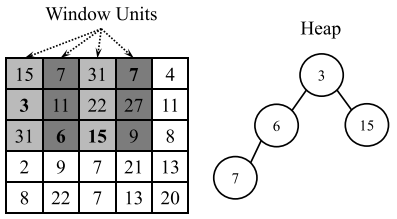


Fig. 6 Initialize the basic window and MIN-heap.

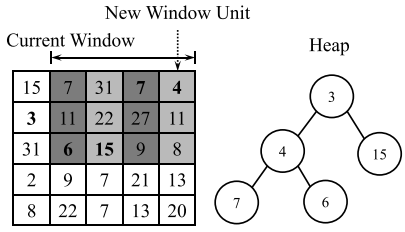


Fig. 7 Process a new unit and update MIN-heap.

3.4 Percentile

3.4.1 Operation Overview

PCTL is an important operator for data analysis in both science [16] and business, such as Facebook [24]. The computing method is explained in the following. For the P-th PCTL ($0 \le P \le 100$) of N elements, it first computes an ordinal rank n as:

$$n = \frac{P}{100} \times N + \frac{1}{2}$$

3.4.2 Incremental Computation Process

For PCTL, we propose application of the self-balancing binary search tree (SBST) [15], [25], which we hereafter express as 'PCTL-SBST.' We take advantage of the close relationship between adjacent windows by reusing many processed data in the previous window. All the redundant expensive sorting works are eliminated, which leads to high efficiency.

The processing steps of the incremental window aggregate for PCTL are similar to the aggregate operators presented in the overview above. We handle the computation process in two stages. The first stage generates basic windows from the first $n-1$ dimensions. Then, for each basic window, we process the second stage. This moves the window forward along with the last dimension and incrementally calculates the PCTL value for each window.

Here, we detail the steps of the incremental computation method with an example. The array is two dimensions and the window size of the query is set to 2×3.

Step 1: Generate a basic window, initialize an SBST, and insert all the values of the basic window into the SBST. **Figure 8** shows this process. The result of a selection operator that computes the n-th smallest key in the SBST is the PCTL value for the basic window.

Step 2: Move the window forward along with the last dimension. Insert the newly arriving elements into SBST. Delete old elements that are no longer present in the current window. After all the updates, the elements inside SBST become exactly the elements of the current window. This is shown in **Fig. 9**.

Step 3: Continue moving forward. Repeat Step 2 to compute the aggregate PCTL values of all the windows derived from the

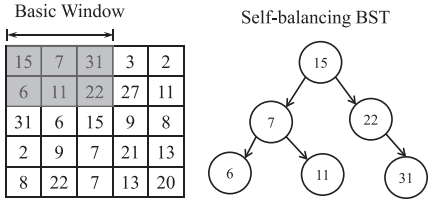


Fig. 8 Basic window and initializing PCTL-SBST.

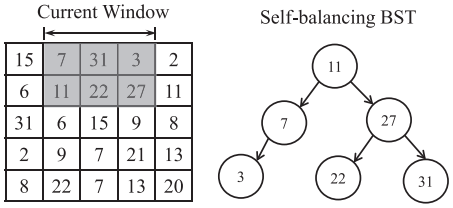


Fig. 9 Update of PCTL-SBST.

basic window.

Step 4: Proceed to a new basic window. Repeat the Steps 1 to 3, which is a compute round. After finishing the compute rounds for all the basic windows, the PCTL window aggregate is completed.

4. Analysis

In this section, we analyze the time complexities for all the aggregate operators thus far discussed with both the naive method and proposed method. In addition, we theoretically verify the improvement enabled by the proposed method. Before the analysis, some parameters in the window aggregate must be defined to elucidate and simplify the description. In a general n-dimensional case, we define the dimension sizes as $D_1 \times D_2 \times \ldots \times D_n$, and the window sizes as $w_1 \times w_2 \times \ldots \times w_n$. Here, D_i represents the array size in the i-th dimension, and w_i represents the window size in the i-th dimension, respectively.

4.1 Naive Method

4.1.1 Sum/Avg/Max/Min

We consider these four aggregate operators together in the same context owing to the necessity of simply scanning all the cells in a window for these operators for the naive method. First, we consider the total number of windows. Each cell in an array has a corresponding window. Therefore, the total number of windows to be computed is the same as the total number of cells in an array. It becomes $\prod_{i=1}^{n} D_i$ in the n-dimensional array case. Then, we consider the scan operation for each window. In each window, there are $\prod_{i=1}^{n} w_i$ cells to be scanned. Therefore, the total time complexity of the naive method for SUM/AVG/MIN/MAX is as follows:

$$O\left(\prod_{i=1}^{n} D_i \prod_{i=1}^{n} w_i\right) \tag{1}$$

4.1.2 PCTL

The quicksort-based method for PCTL is herein handled as the naive method for its high computational cost. It separately computes each window. For each window, an execution of quicksort is needed instead of a simple scan. As analyzed above, the total number of windows to be computed is $\prod_{i=1}^{n} D_i$. Then, we consider the quicksort part. For each window, an execution of quicksort must be processed. It sorts all the elements in the window. It is known that sorting N elements with quicksort costs $O(N \log N)$ on average. Because there are $\prod_{i=1}^{n} w_i$ cells in a window, the time complexity for computing one window becomes $O\left(\prod_{i=1}^{n} w_i \log \prod_{i=1}^{n} w_i\right)$. From the analysis above, the total time complexity for the quicksort method becomes:

$$O\left(\prod_{i=1}^{n} D_i \prod_{i=1}^{n} w_i \log \prod_{i=1}^{n} w_i\right) \tag{2}$$

4.2 Proposed Method

We now analyze the time complexity of incremental computation methods. Different data structures are used to maintain the intermediate values. Therefore, the analyses are separately described.

4.2.1 Summation and Average

First, we consider the number of basic windows. Basic windows are determined by the first $n-1$ dimensions. Therefore, the total number of basic windows is $\prod_{i=1}^{n-1} D_i$. In each compute round of the basic window, the windows derived by D_n are to be computed because the moving process is along with the last dimension. For each derivative window, the new window unit to be scanned has $\prod_{i=1}^{n-1} w_i$ cells. Therefore, the time complexity is:

$$O\left(\prod_{i=1}^{n} D_i \prod_{i=1}^{n-1} w_i\right) \tag{3}$$

Compared to the complexity of the naive method shown in Eq. (1), the proposed method reduces multiply factor w_n in the time complexity. w_n denotes the window size in the last dimension of an array. In theory, this means that the proposed method obtains a speedup factor proportional to w_n compared with the naive method.

From the expression above, we additionally identify a notable feature of the proposed method. Specifically, the time complexity of the proposed method for window SUM/AVG aggregates has no relationship with window size w_n. This means that, no matter how large the window is on the last dimension, it does not affect the efficiency of the proposed method. A similar feature also exists in MIN, MAX, and PCTL operators, as shown below.

4.2.2 Maximum and Minimum

We now analyze the MAX/MIN operators of the proposed method. In addition to the operation of the window unit scan, the part of the heap operators must also be considered. We analyze the time complexity in two aspects. The first aspect is the MIN/MAX calculation of the window units; the second aspect involves operations in the heap. The calculation of window units is similar to SUM/AVG. The number of cells that must be scanned is the same. Therefore, the time complexity of this part is:

$$O\left(\prod_{i=1}^{n} D_i \prod_{i=1}^{n-1} w_i\right) \tag{4}$$

We consider the heap operations part. The heap is a well-known data structure. To insert a new node, it costs $O(\log L)$. To delete the root node and maintain the remaining nodes, it costs $O(\log L)$. L represents the number of nodes in the heap.

In terms of analysis of the window query, there are $\prod_{i=1}^{n-1} D_i$ basic windows. For each basic window, a total D_n MIN/MAX values of window units must be inserted into the heap. Among these values, at most $D_n - w_n$ ones are deleted from the heap during the incremental computation process. Therefore, we take the size of the heap as D_n, and the number of insertions and deletion operations can also be approximated as D_n. Therefore, the time complexity of the heap part becomes:

$$O\left(\prod_{i=1}^{n-1} D_i D_n \left(\log D_n + \log D_n\right)\right) = O\left(\prod_{i=1}^{n} D_i \log D_n\right) \quad (5)$$

Adding these two parts for the window aggregates of MAX/MIN in Expressions (4) and (5), the time complexity of the proposed method is:

$$O\left(\prod_{i=1}^{n} D_i \left(\prod_{i=1}^{n-1} w_i + \log D_n\right)\right) \quad (6)$$

In this expression, the item $\log D_n$ is much smaller than $\prod_{i=1}^{n-1} w_i$ in most cases. Therefore, the time complexity of MAX/MIN is only slightly larger than SUM/AVG. Similar to the SUM/AVG case, the time complexity is unrelated to the size of the last dimension in the queried window, which is w_n.

4.2.3 PCTL

We first consider a 2D array case with dimension sizes of $X \times Y$. The window size is set as $a \times b$. In this case, a total of X basic windows exist. According to the process introduced in Section 3.3, in each compute round for a basic window, we must move on the second dimension (with size Y) to obtain new windows and process the incremental computation. Therefore, each basic window has Y derivative windows to be calculated.

Here we explain more details on how to compute the selection in self-balancing BST efficiently. As introduced above, in order to obtain the percentile value, it is necessary to find the n-th smallest key in the BST. However, standard BST only supports searching on specified keys. Our solution is to maintain the rank of each node. For each node, an extra value representing the size of the subtree with the node as a root is maintained, referred to as S. This value is maintained during any operation of BST with only cost $O(1)$, simply adding up the S values of left subtree, right subtree and one as the node itself. Considering a node, its rank in the tree is easy to compute, equals to its left child's S value plus 1. Because the node is larger than any nodes from its left subtree and smaller than any nodes from its right subtree, which is the nature of BST. Therefore, with S values maintained, the rank of each node is clear, and the selection can be achieved in a similar way as how normal key-search operation of BST does. The only difference is that we search for a specific rank instead of searching a specific key. The complexity is also the height of the tree, which is $\log N$ in a self-balancing BST with N nodes.

When calculating each derivative window, elements in one old window unit must be deleted from SBST, and elements in the new window unit must be inserted. In this specific 2D case, a window unit contains a cells, as shown in **Fig. 10**. Then, one selection is executed to obtain the PCTL result of the current window. Meanwhile, the SBST always maintains exactly all the elements of the current window. The size of SBST is $a \times b$. Therefore, every single operation in the SBST costs $\log ab$.

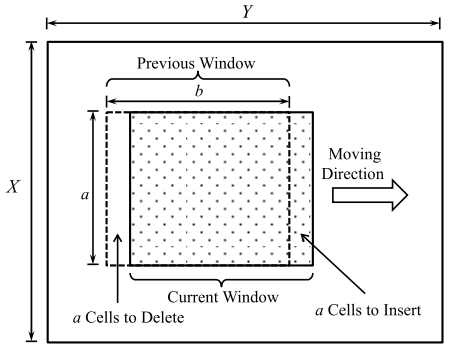


Fig. 10 Details of window in a one-step move. 2D array with size $X \times Y$ and window size of $a \times b$.

From the analysis above, for each movement step, we obtain the time complexity by adding the costs for the insertions, deletions, and selections; i.e., $O(a \log ab + a \log ab + \log ab)$. It can be simplified as $O(a \log ab)$. Therefore, the total time complexity of the 2D case is $O(XYa \log ab)$. With the same definition of dimension and window sizes in the 2D case, the complexity of the quicksort method is $O(XYab \log ab)$. It is obvious that our proposed method has a speedup by a factor of b compared with the quicksort method.

For an n-dimensional array, the analysis is similar. First, we consider the number of basic windows, which is $\prod_{i=1}^{n-1} D_i$. This is the same as other aggregate operators. Then, we consider each basic window. In the compute round based on one basic window, the window moves along with the last dimension. Therefore, D_n moving steps exist, and D_n derivative windows must be computed.

For each derivative window, the number of new cells to be inserted into SBST is $\prod_{i=1}^{n-1} w_i$. The same number of cells is to be removed. Then, one selection is executed to obtain the PCTL result. When executing all of these SBST operations, the size of the tree is the same as the total size of the windows, which is $\prod_{i=1}^{n} w_i$. Therefore, each single operation of SBST costs $\log(\prod_{i=1}^{n} w_i)$. There are $\prod_{i=1}^{n-1} w_i$ times of insertions, $\prod_{i=1}^{n-1} w_i$ times of deletions, and one time of selection. All these operations in SBST must be executed in each window. To summarize them, the complexity of the proposed method for one window is:

$$O\left(\left(\prod_{i=1}^{n-1} w_i + \prod_{i=1}^{n-1} w_i + 1\right) \log \prod_{i=1}^{n} w_i\right) \quad (7)$$

It can be simplified as:

$$O\left(\prod_{i=1}^{n-1} w_i \log \prod_{i=1}^{n} w_i\right) \quad (8)$$

As analyzed above, $\prod_{i=1}^{n-1} D_i$ basic windows exist. For each basic window, D_n windows are to be computed. In total, the time complexity for our proposed method becomes:

$$O\left(\prod_{i=1}^{n} D_i \prod_{i=1}^{n-1} w_i \log \prod_{i=1}^{n} w_i\right) \quad (9)$$

Comparing Eqs. (9) and (2), the latter of which expresses the complexity of the quicksort method, the proposed method obtains a speedup by a factor of w_n in time complexity. This means that, in theory, the proposed method obtains a speedup factor proportional to w_n compared with the quicksort method.

5. Design and Implementation

5.1 Plugin Mechanism

To evaluate the performances of the proposed method and naive method, we implement the proposed method on SciDB. A

straightforward method to implement the new operator with high efficiency is to add a new built-in operator into SciDB. 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 using the C++ language through the plugin mechanisms. The implemented plug-in operators are stored in the SciDB library system. Then, the implemented plugin operators can be loaded in the SciDB library system. Interestingly, following this mechanism, once the plugin library is loaded into the system, the operator can perform in the similar way to the built-in operators in SciDB for data accessing, processing, and operator-result fetching. Therefore, the use of the 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 'Logical_MyOperator.cpp' and 'Physical_MyOperator.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 of array schemas of the new operator. The 'Physical' file is the body, which describes data processing details that generate the result of the operator. Moreover, 'plugin.h' should be included when building the UDO to notify SciDB that the code is UDO.

A large array in SciDB at times does not fit into physical memory. In SciDB, such a large array is divided into small chunks and distributed within a cluster. A chunk is a sub-array whose data can fit in memory. Operators can be efficiently executed chunk by chunk without extra disk I/O. Computing the result while iterating the chunks in the resulting array is the most important part of the implementation. The function is referred to as 'calculateNextValue().' When the resulting array of the window aggregate iterates into a new cell, SciDB calculates the value for the cell using this function. We show the implementation in the form of pseudo-code in Section 5.2.

5.2 Pseudocode

To clarify the details of our implementation, we show pseudocode of 'calculateNextValue()' for both the naive and proposed methods. All of the actual code is available in GitHub [17], [18].

Pseudocode 1 describes the UDO implementation of the naive method. Each time the resulting array iterates to a new cell, the function 'calculateNextValue()' shown in psudocode 1 is invoked to compute the aggregate value of the window corresponding to the cell (lines 5–6). The 'aggregator' is a class that computes the aggregate value of a set of data. It supports 'insert,' which can feed one new element into the set, and 'accumulate,' which can return the aggregate result of the current dataset, including all elements inserted so far.

In addition, '_inputChunk' is a built-in class provided by SciDB. It manages the chunks that are processed by an operator. The chunk data required for data processing are automatically fetched into the memory if they are in storage. Therefore, users are not required to explicitly focus on storage access with

Pseudocode 1: calculateNextValue() in the naive method

```
// Get the current window area scope
 1: FOR i ← 1 TO nDim DO
 2:     firstWinPos[i] ← currPos[i] – winFirst[i];
 3:     lastWinPos[i] ← currPos[i] + winLast[i];
 4: aggregator.clear();
 5: FOR (cell c ∈ window(firstWinPos,lastWinPos)) DO
 6:     aggregator.insert(_inputChunk.accessData(cell c));
 7: nextValue ← aggregator.accumulate();
```

Pseudocode 2: calculateNextValue() in the proposed method

```
// Get the current window area scope
 1: FOR i ← 1 TO nDim DO
 2:   firstWinPos[i] ← currPos[i] – winFirst[i];
 3:   lastWinPos[i] ← currPos[i] + winLast[i];
 4: IF (current window is a basic window) DO
 5:   FOR i ← firstWinPos[nDim] TO lastWinPos[nDim]
DO
 6:     bufTool.insert(calcWindowUnit(i));
 7:   ELSE DO
 8:     bufTool.remove();
 9:     bufTool.insert(calcWindowUnit(lastWinPos[nDim]);
10:     nextValue ← bufTool.getCurrentValue();
```

Pseudocode 3: calcWindowUnit() in the proposed method

```
1:  aggregator.clear();
2:  FOR (cell c ∈ windowUnit(lastDimValue)) DO
3:      aggregator.insert(_inputChunk.accessData(cell c));
4:  RETURN aggregator.accumulate();
```

the class. It supports the 'accessData' method to obtain the data of one cell based on its multi-dimensional coordinates. When the resulting array iterates into a new cell, the scope of the window area centered on this cell is computed first. Then, all of the cells inside this window scope are scanned and inserted into an aggregator. They are finally accumulated into the aggregate result of the current window.

Pseudocode 2 describes the UDO for proposed method. In the code, 'bufTool' is our own class. The bufTool provides a simple abstraction for different aggregate operators. Different data structures are selected to efficiently reuse the intermediate results of previous windows. To simplify the code, the data manipulation methods are unified through bufTool. The bufTool class supports the 'insert' method, which inserts the aggregate result of the new window unit, and the 'remove' method, which removes the aggregate result of the oldest window unit that logically no longer exists in the current window. In the proposed method, when the resulting array iterates into a new cell, only one window unit is needed to be scanned, whereas the whole window scan is needed for the naive method. A window unit is one slice of the window when the position of the last dimension fixed. The first $n-1$ dimension positions vary in the scope of the window, which is shown as 'firstWinPos, lastWinPos' in the pseudocode.

Pseudocode 3 describes 'calcWindowUnit,' which is invoked from 'bulTool.insert.'

6. Experiment

6.1 Environment

To evaluate the performances of the proposed method and naive method, we implemented all five aggregate operators herein

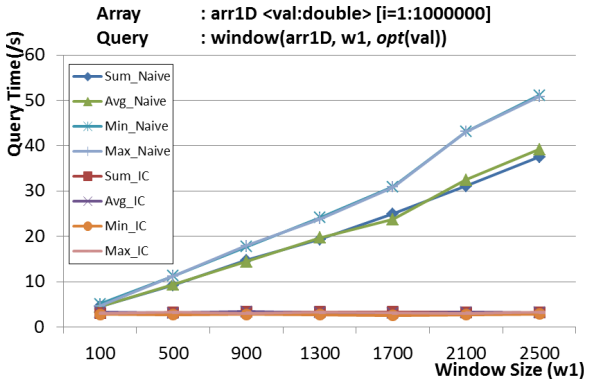


Fig. 11 Query execution time for array 'arr1D' (incremental computation = IC).

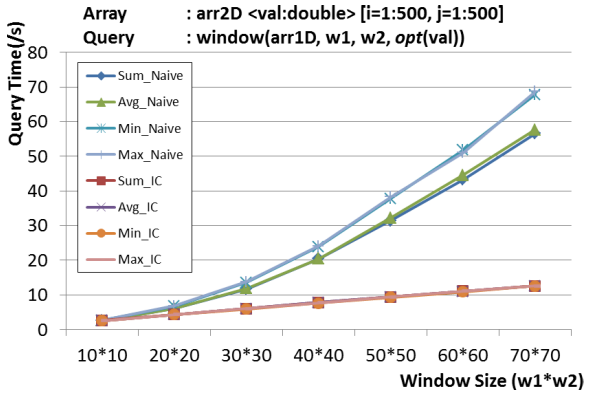


Fig. 12 Query execution time for array 'arr2D' (incremental computation = IC stands).

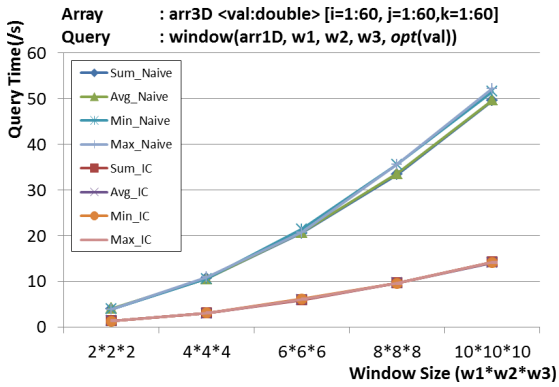


Fig. 13 Query execution time for array 'arr3D' (incremental computation = IC).

introduced into SciDB. In addition, we implemented the quicksort method for PCTL because the window aggregate of PCTL is not supported by SciDB. For the implementation, we employed the C++ language; approximately 2,300 lines; and SciDB version 13.12. All source code of our implementation is available on GitHub [17], [18]. From the code, only plugin files were built. SciDB supports a plugin mechanism that allows users to implement their own operators. The operator must be loaded into SciDB as a plugin. Regarding the machine environment, we set the configuration parameters as follows: CentOS 6.5 operating system; Intel(R) Xeon(R) E5620 2.40-GHz CPU; and a main memory size of 24 GB.

6.2 Max/Min and Sum/Avg

6.2.1 Micro benchmark with Synthetic Workload

For MAX, MIN, SUM and AVG, three series of experiments were conducted with different dimensional settings. We synthetically generated three arrays—'arr1D,' 'arr2D,' and 'arr3D'—with different dimensional numbers, as named. They each contained one attribute named 'val.' The value was randomly generated with a range of [0, 1,000,000]. For each array, we designed the query parameters with window sizes in each dimension linearly increasing to investigate how window size affects the query execution time.

The results are shown in **Figs. 11**, **12**, **13**. In the figures, 'opt_naive' represents the naive implementation of operators in SciDB, which can be SUM/AVG/MIN/MAX. 'opt_IC' stands for our proposed method.

Figure 11 shows the results of a one-dimensional (1D) case. The array 'arr1D' has a size of 1,000,000, and the window size of the query increases from 100 to 2,500. The result shows improvement in execution time of the proposed method, especially when the window size is large. The highest two lines are naive MIN and naive MAX, followed by the SUM and naive AVG. The lowest four lines represent the proposed methods for these operators. Because the execution times of proposed methods are almost the same, and they are much smaller than that of the naive ones, these four lines almost overlap into a single one. The maximum performance improvement is by a factor of 17.9 with a window size of 2,500 for the MIN operator and 12.5 for the SUM operator.

Figure 12 shows the result of the 2D case. The array 'arr2D' has a size of 500×500, and the window size of the query is increased in both dimensions. Similar to the 1D case, the highest two lines are naive MIN and naive MAX, followed by naive SUM and naive AVG. The lowest four lines represent the proposed methods for these operators. Again, the results clearly show the improvement of the proposed methods.

Figure 13 shows the results of the experiment for the 3D case. The 'arr3D' has a size of $60 \times 60 \times 60$, and the window sizes in each dimension linearly increase. The top four lines represent naive methods; the bottom four lines represent the proposed method. When the processing window aggregated the query with the proposed method, the workload was almost the same and the execution costs had similar running times regardless of the detailed operator. Therefore, the four lines of the proposed methods almost overlap into a single line in the figures.

From the result, another feature of the proposed method is evident. Specifically, the execution time of the method is not related with the window size in the last dimension. This result is consistent with the time complexity analysis presented in Section 4.2. In the time complexity expression of the incremental computation method, w_n does not exist, as shown in Eqs. (3) and (6). Therefore, w_n would not affect the execution time. In Fig. 11, regardless of the extent to which w_1 increases, the execution time of the proposed method remains almost the same. In Fig. 12, the execution time of the proposed method shows linear growth according to w_1 instead of quadratic growth as $w_1 \times w_2$. The same feature is found in Fig. 13, which shows the 3D case. From the results, it is again evident that our time complexity analysis is consistent with the experimental results.

6.3 PCTL

6.3.1 Benchmark with JRA-55 dataset

To compare the efficiency of the naive method and our proposed method for the PCTL operator, we designed two series of experiment cases with different parameter settings. One was aligned to a real analysis situation; the other one was a synthetic situation. In both cases, we used a real dataset, JRA-55.

6.3.1.1 JRA-55 Dataset

The JRA-55 [14] dataset (named for the Japanese 55-Year Reanalysis) dates from January 1, 1958 to December 31, 2012. This 55-year range coincides with the establishment of the global Radiosonde Observing System. It covers a global area and supports a regular latitude–longitude Gaussian grid (145 latitude by 288 longitude, nominally 1.25 degrees), and daily six-hour data [13]. The data have three dimensions—longitude, latitude, and time—and are managed by array databases, which makes them an appropriate choice for evaluating the performance of our proposed method.

In the experiment, the selected data attributes and query parameter settings were applied to real meteorological applications that require computing PCTL followed by meteorological research [16]. For the experiment, we loaded surface temperature data of JRA-55 for 20 years into SciDB. The size of the surface temperature data for one year was approximately 4 GB; for 20 years, it was a total of 80 GB. When loading the data, we converted the JRA-55 files from GRIB2 format to CSV format because SciDB-13.12 does not provide a mechanism to load the former format, whereas CSV is provided.

6.3.1.2 Evaluation with a Real Workload

The first evaluation series was over an array containing surface temperature data from 2012 in the JRA-55 dataset. The evaluated queries are required by meteorologists in real analyses [16]. The array was 3D with a size of $288 \times 145 \times 366$ respectively corresponding with longitude, latitude, and time. The first two dimensions specified the location of the area on the earth, while the last dimension specified the temporal nature. In each cell, the main attribute was the surface temperature at 12:00 on the given day in the JST timezone.

The window aggregate query herein evaluated was required by our cooperating meteorologists [16]. They had to calculate the PCTL values over sliding temporal windows. More specifically, the request needed to calculate PCTL of temperature data in 30-day increments. Meanwhile, on the spatial aspect, they did not require any overlapping windows, and computation was required only over single spatial cells. The situation is shown in **Fig. 14**.

When designing the window parameters of this application, we set the window size of the first two dimensions as 1×1, and we increased only the window size in the temporal dimension. The used attribute was the surface temperature at noon and three PCTL percentages were evaluated. They were 25%, 50%, and 75%.

The results are shown in **Fig. 15**. As explained above, for both methods, different lines with different percentage settings almost overlap in a single line. For both the naive sorting method and the proposed method, different values of percentages would not change the workload of the algorithm; moreover, the execution

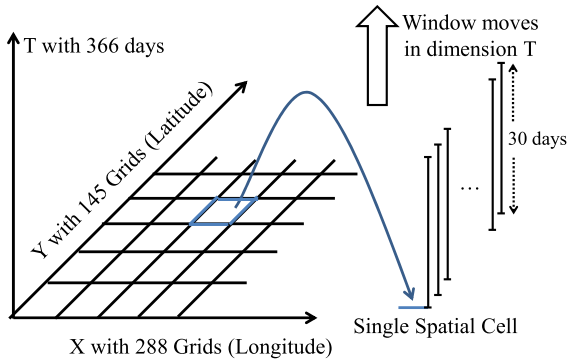


Fig. 14 Window query in a real meteorological application computing PCTL in a 30-day window.

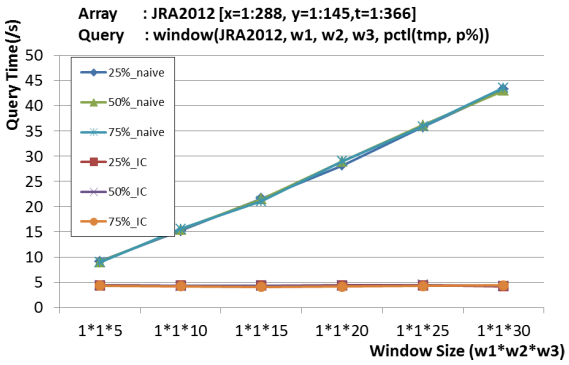


Fig. 15 Query execution time with window size (incremental computation = IC).

times were the same when only changing the PCTL percentage maintained the other parameters as fixed.

The results show improvements in the running time of the proposed method against the naive sorting method. As the window size increased, the effect of improvement likewise increased. The last case with a window size of $1 \times 1 \times 30$ was exactly the query needed in the meteorological analysis required by meteorologists [16]. In this case, we observed the maximum improvement, which occurred by a factor of 10.2.

From the result, the interesting performance feature of the incremental computation method was again identified. With all the other parameters fixed, regardless of the extent to which w_n (the window size in the last dimension) varied, the processing time of the query almost remained the same. This feature is consistent with our time complexity analysis described in Eq. (9).

6.3.1.3 Evaluation with a Synthetic Workload

We used the second series of the experimental dataset. In this case, we only changed the window sizes in the first two dimensions (longitude and latitude) and retained the same time dimension. When the window size increased, the naive quicksort method incurred a longer time. Therefore, we selected a smaller array than the one used in the previous experiment. It included data of only one month, the JRA-55 data of January 2012. This array contained temperature at 00:00, 6:00, 12:00, and 18:00 in one day instead of only the data at 12:00. Therefore, the number of data items for the third dimension of this array was 124 (31×4) instead of 31.

The results of the experiment are shown in **Fig. 16**. Compared with the quicksort method, the proposed method has a speedup

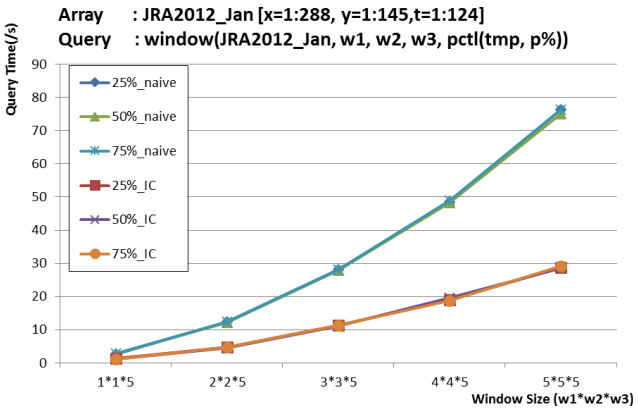


Fig. 16 Query execution time with window size (incremental computation = IC).

by a factor of w_n. The experimental result is consistent with our analysis in Section 4. In the second experiment series, as the w_3 remained the same, the speedup value of the proposed method against the quicksort method remained almost the same. Meanwhile, in the first experiment series, as w_3 linearly increased, so did the speedup value. To more clearly show this, we divided the speedup values with w_3 in each evaluation case of both series. The results are both very close to the same constant. This demonstrates that our analysis of the speedup factor as w_3 is appropriate.

7. Related Work

The window aggregate is an important class of operators whose acceleration has been well studied. SAGA [12] presents efficient approaches for structural aggregates of array data. SAGA deals with multi-dimensional data and the window aggregate, while focusing on reducing disk I/O cost. Unlike our work, reuse of intermediate results or incremental computation is out of scope in SAGA. Incremental computation has been studied in the context of stream data processing [20], [22], [35]. While their performance improvement is quite meaningful, their focus on 1D data and multi-dimensional problems are out of scope.

For temporal aggregates of interval data, important works exist, such as balanced-tree [10] and SB-tree [11], to incrementally compute the query. The difference between these works and ours is that the underlying data structures used are different. The works introduced above are designed to deal with particular types of data, specifically interval data and stream 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. Diamond, hexagon, and polygonal shaped window aggregates are discussed in Refs. [26], [27]; however, such shapes are not supported by any array database systems, including SciDB. The convolution filter exploits incremental computation and is implemented in OpenCV [28]. Nevertheless, it does not address large arrays, which do not fit into memory; moreover, it too is not supported by SciDB. Some contest problems [29], [30] at ACM ICPC are more complicated than our problem, although they completely differ from ours. Furthermore, all the above works focus only on algorithms and lack a system design perspective. In this paper, on the other hand, we provide design and implementation on SciDB.

8. Conclusions

In this paper, we proposed efficient algorithms for window aggregate operators in array databases based on an incremental computation scheme. We developed acceleration techniques for five types of operators that exploit different types of data structures to achieve efficiencies: list for summation and average, heap for maximum and minimum, and balanced binary search tree for percentile.

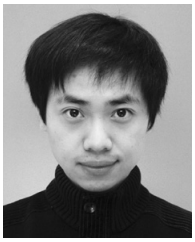
To evaluate the proposed method, we presented time complexity analyses for the proposed method and naive method. We implemented the proposed method for five operators in a real array database, SciDB. The results of experiments on SciDB showed that improvement by the proposed method for minimum, summation, and percentile operators were by a factor of 17.9, 12.5, and 10.2, respectively. Our proposed method therefore accelerates window aggregates on the SciDB real array database system.

Acknowledgments This work is partially supported by KAKENHI (25280043HA), JST CREST "System Software for Post Petascale Data Intensive Science", JST CREST "Extreme Big Data Next Generation Big Data Infrastructure Technologies towards Yottabyte/Year" and JST CREST "Statistical Computational Cosmology with Big Astronomical Imaging Data". We also appreciate Prof. Mio Matsueda for his fruitful comments and discussions with meteorological applications.

References

[1] Seering, A., Cudré-Mauroux, P., Madden, S. and Stonebraker, M.: Efficient Versioning for Scientific Array Databases, *ICDE*, pp.1013–1024 (2012).
[2] Brown, P.G.: Overview of SciDB, Large Scale Array Storage, Processing and Analysis, *SIGMOD*, pp.963–968 (2010).
[3] Soroush, E., Balazinska, M. and Wang, D.: ArrayStore: A Storage Manager for Complex Parallel Array Processing, *SIGMOD*, pp.253–264 (2011).
[4] Soroush, E. and Balazinska, M.: Time Travel in a Scientific Array Database, *ICDE*, pp.98–109 (2013).
[5] 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* (2009).
[6] Kersten, M.L., Zhang, Y., Ivanova, M. and Nes, N.: SciQL, A query language for science applications, *EDBT/ICDT Workshop on Array Databases* (*AD*), pp.1–12 (2011).
[7] Ballegooij, A.V., Cornacchia, R., deVries, A.P. and Kersten, M.: Distribution Rules for Array Database Queries, *DEXA*, pp.55–64 (2005).
[8] Zhang, Y., Kersten, M. and Ivanova, M.: SciQL: Bridging the gap between science and relational DBMS, *IDEAS*, pp.124–133 (2011).
[9] Baumann, P., Dehmel, A., Furtado, P., Ritsch, R. and Widmann, N.: The Multidimensional Database System RasDaMan, *SIGMOD*, pp.575–577 (1998).
[10] Moon, B., Fernando, I., López, V. and Immanuel, V.: Scalable Algorithms for Large Temporal Aggregation, *ICDE*, pp.145–154 (2000).
[11] Yang, J. and Widom, J.: Incremental computation and maintenance of temporal aggregates, *VLDB J.*, Vol.12, No.3, pp.262–283 (2003).
[12] Wang, Y., Nandi, A. and Agrawal, G.: SAGA: array storage as a DB with support for structural aggregations, *SSDBM*, Article No.9 (2014).
[13] Ebita, A. et al.: The Japanese 55-year Reanalysis "JRA-55": An Interim Report, *SOLA*, Vol.7, pp.149–152 (2011).
[14] JRA-55 Data Archive, available from ⟨http://gpvjma.ccs.hpcc.jp/~jra55/#⟩.
[15] Sleator, D.D. and Tarjan, R.E.: Self-Adjusting Binary Search Tree, *Journal of ACM*, Vol.32, No.3, pp.652–686 (1985).
[16] Matsueda, M. and Nakazawa, T.: Early warning products for severe weather events derived from operational medium-range ensemble forecasts, *Meteorol. Appl.* (2014).
[17] Jiang, L.: Source codes of the Incremental Window Percentile, available from ⟨https://github.com/ljiangjl/Percentile-in-SciDB.git⟩.

[18] Jiang, L.: Source codes of the Incremental Window Sum/Avg/Max/Min, available from ⟨https://github.com/ljiangjl/IC_Window.git⟩.
[19] VanderPlas, J., Soroush, E., et al.: 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).
[20] 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, *SIGMOD Rec.*, Vol.34, No.1 (2005).
[21] Jiang, L., Kawashima, H. and Tatebe, O.: Incremental window aggregates over array database, *IEEE BigData*, pp.183–188 (2014).
[22] Wu, Y., Maier, D. and Tan, K.-L.: Grand challenge: SPRINT stream processing engine as a solution, *DEBS*, pp.301–306 (2013).
[23] Ceri, S. and Widom, J.: Deriving Production Rules for Incremental View Maintenance, *VLDB*, pp.577–589 (1991).
[24] Presto query engine, available from ⟨https://prestodb.io/docs/current/functions/aggregate.html⟩.
[25] Martínez, C. and Roura, S.: Randomized Binary Search Trees, *J. ACM*, Vol.45, No.2, pp.288–323 (1998).
[26] Sun, C.: Moving average algorithms for diamond, hexagon, and general polygonal shaped window operations, *Pattern Recognition Letters*, Vol.27, No.6, pp.556–566, Elsevier (2006).
[27] Sun, C.: Diamond, Hexagon, and General Polygonal Shaped Window Smoothing, *DICTA*, pp.39–48 (2003).
[28] Normalized Box Filter, OpenCV Tutorials.
[29] Maximum Sum: UVA Online Judge, available from ⟨https://uva.onlinejudge.org/external/1/108.html⟩.
[30] Square Carpets, ACM ICPC, Japan Domestic (Oct. 2003), available from ⟨http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=1128⟩.
[31] The Large Hadron Collider, available from ⟨http://home.cern/topics/large-hadron-collider⟩.
[32] Subaru Telescope, available from ⟨http://subarutelescope.org/⟩.
[33] Atzori, L., Iera, A. and Morabito, G.: The Internet of Things: A survey, *Computer Networks: The International Journal of Computer and Telecommunications*, Vol.54, No.15, pp.2787–2805 (2010).
[34] Modis Benchmark, available from ⟨http://people.csail.mit.edu/jennie/elasticity_benchmarks.html⟩.
[35] Tangwongsan, K., Hirzel, M., Schneider, S. and Wu, K.-L.: General Incremental Sliding-Window Aggregation, *PVLDB*, Vol.8, No.7, pp.702–713 (2015).
[36] Brown, P.: A Survey of Scientific Applications using SciDB, *New England Database Day Program* (2015).
[37] Stonebraker, M., Duggan, J., Battle, L. and Papaemmanouil, O.: SciDB DBMS Research at M.I.T, *IEEE Data Eng. Bull*, Vol.36, No.4, pp.21–30 (2013).
[38] Rogers, Y.: Moving on from weiser's vision of calm computing: Engaging ubicomp experiences, *UbiComp*, pp.404–421 (2006).



Li Jiang received his B.S. in Computer Science from Zhejiang University in 2013, the M.S. in Computer Science from University of Tsukuba in 2016. He is a software engineer in Google Dublin.

Hideyuki Kawashima received his Ph.D. from Science for Open and Environmental Systems Graduate School of Keio University, Japan. He was a research associate at Department of Science and Engineering, Keio University from 2005 to 2007. From 2007 to 2015, he was an assistant professor in University of Tsukuba. He is now an associate professor in Center for Computational Sciences, University of Tsukuba.

Osamu Tatebe received his Ph.D. in computer science (1997, Univ. of Tokyo) for his work on Parallel Iterative Solver. He worked at Electrotechnical Laboratory (ETL), and National Institute of Advanced Industrial Science and Technology (AIST) until 2006, becoming a chief research scientist. He is now a professor in Center for Computational Sciences at University of Tsukuba. His research area is high-performance computing, data-intensive computing, and parallel and distributed system software.