Implementing Incremental Aggregate Computation on SciDB

Li Jiang[†1] Hideyuki Kawashima[†2] Osamu Tatebe[†3]

SciDB is a typical array DBMSs, a new kind of databases dealing with big data storing and processing in science fields. The window aggregation is a typical aggregate query of it, especially with the percentile function, which is useful and frequently used. However, generally naive sorting method is used to calculate the percentile window aggregates, which will do much redundant computation and lead to low efficiency. In this thesis, we propose an improved method with incremental computation for percentile window aggregate queries. It uses data structure of self-balancing binary search tree to maintain needed data and reuses them when computing, eliminates the redundant computation. The time complexity analysis is offered, which shows the advantages of the proposed method clearly. We implement this method in SciDB, run performance experiments and also examine the method's parallel processing features.

1. Introduction

Nowadays, science and industry are growing increasingly data-intensive, efficient analysis of big data is getting more and more important. In many fields, data has multiple dimensions and does not fit in the table data model so well, leading a high cost in some analysis tasks. In order to efficiently store and analyze such multi-dimensional data, array database systems appeared, with array as basic data model instead of table.

Our work is to improve the performance of an important query in array DBMSs, the window aggregates. By exploiting the idea of incremental computation, redundant works are reduced and performance is highly improved. In this paper, we focus on an aggregate operator "percentile", which is useful for scientist such as meteorologists. Since the function is complicated, its acceleration is not trivial. Improvement of other aggregate operators can be found in our previous work [12].

The proposed method, as well as the naive one, is implemented into SciDB [7] which is the most popular array database system. In the experiment, we used JRA55 dataset [14], which is a series of meteorological data, to evaluate the performance of proposed method in a real application.

SciDB

SciDB is a typical array database system, designed to store and manipulate big array data that often seen in science community. The data model is multi-dimensional array instead of table, which is different from relational database. The array data model naturally fits in data schema well in many fields of science such as meteorology [16] and astronomy [18, 19]. SciDB inherently supports efficient complex analyses over multi-dimensional data.

On the viewpoint of system architecture, SciDB adopts a design of shared nothing storage architecture, and it can process a query in parallel. For storing a large array, SciDB divides it into small chunks and distribute these chunks across multiple servers in a cluster, which makes it possible to compute a query in parallel for each chunk.

Window Aggregates

Window aggregates are one of the most important operators in array databases. It's a type of operator frequently used when analyzing multi-dimensional array data, as quite a common and necessary query in array databases or array data processing systems.

A window query computes aggregate operator over a moving window. Here, a window is more like a sub-array of the original array and its sizes in each dimension are specified by users. The window starts at the first grid of array and moves in stride-major order from the lowest to highest value in each dimension. Figure 1 shows a 3x3 sized window aggregate in 2-dimension, and how the window moves. The result of a window query is an array with the same size as the input array, with each cell containing the aggregate result of that cell's corresponding window in the original array.

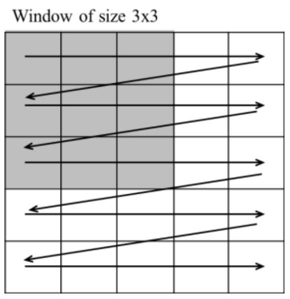

Figure 1 Window aggregate in a 2-dimensional array

Percentile Operator

A percentile is a measure used in statistics indicating the value below which a given percentage of observations fall. For example, among all scores of a class, if a score is in 85th percentile, then it is exactly higher than 85% of all the scores [17].

Percentile has many real applications in practice. It is very

†1 Graduate School of System and Information Engineering, University of Tsukuba
†2 Faculty of Engineering, Information and System, and Center for Computational Sciences, University of Tsukuba
†3 CREST, JST

useful and can provide interesting information when analyzing data. For example, meteorologists want to compute percentiles over moving time windows at the scope of global areas [16]. This exactly fits in a window aggregate query in array database, with the aggregate function to be percentile.

Generally, percentile is computed by sorting. But in the case of window aggregate, it requires to process one sort for each window, which would be time consuming since there are so many windows. Thus, we propose another method to process the window aggregate of percentile more efficiently by reusing previous window's data with the concept of incremental computation. Details of this method are presented in Section 3.

JRA55 data

Conducted by the Japan Meteorological Agency (JMA), JRA55 is the second Japanese global atmospheric reanalysis project. It covers 55 years, extending back to 1958, coinciding with the establishment of the global radiosonde observing system. This data project covers the global area, supports a regular latitude-longitude Gaussian grid (145 latitudes by 288 longitudes, nominally 1.25 degree), and daily 6-hourly data [13]. So this is a typical multi-dimensional data, exactly 3 dimensions: the longitude, latitude and time. Such data can be managed and analyzed by array databases efficiently, making it a nice choice for testing the performance of our proposed method against naive method.

2. Related Works

2.1 Related researches

Because array database is quite a new type of database, as a typical query of it, there are not so many works related about accelerating window aggregate.

About array databases, there are some researches to extend some scientific features, such as data versioning [1, 4] and uncertain data [9]. Also, efficient distributed storage and parallel processing of arrays are discussed in some works [2, 3].

As the most popular array DBMS, SciDB gains more attentions [5, 7, 8], these works focus on the architecture design and low-level array storage, while our work is to improve a specific query in array database. There is another array database, SciQL[6, 11], which is not so mature as SciDB.

2.2 Previous works

In previous works, we improved some other aggregate operators of window aggregates with incremental computation, such as average, minimum and maximum. Compared with percentile, these functions are simple and easier to adapt incremental computing. The experiment results showed a great improvement in performance [12].

Also, in this previous work, we implemented an array data processing system, which can only process analytical tasks with arrays in main memory. The system is simply designed, lacks of data management features and can only run queries in one thread.

In this paper, we choose to implement the proposed method into SciDB, a mature open-source array database system. The result would be more convincing than our previous experimental environment, and the real workload of our method can be showed in a well-constructed array database system.

3. Method Details

This Section discuss about the details of the methods for computing the window aggregate of percentile. First, for a set of data observations, how would percentile be computed would be introduce with details. Then we introduce the naive method, the quicksort based method and then the proposed method that leverages incremental computing using data structure, *self-balancing binary search tree*.

3.1 Percentile

In fact, there is no standard definition of how to compute percentile, however it is known that all the definitions would get similar results, especially when the number of observation is very large [17].

The definition we selected is referred to as nearest rank. The computing method is explained in the following. For the P-th percentile $(0 \leq P \leq 100)$ of N values, it first computes an ordinal rank n as

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

Then, it rounds the result to the nearest integer, then the value corresponding the rank n among all N values (arranged from least to greatest) is obtained as the percentile value.

3.2 Non-incremental computation

Because the calculation of the percentile can be treated as computing the rank n-th value in the data set, a natural way would be sorting the data set first, then the n-th smallest value is obviously computed. In this way, quicksort would be a general choice for sorting the data, with time complexity as $O(N\log N)$, in which N is the total number of sorted values. As for the percentile window aggregate, one sort process needed for computing each one window. We refer this method to "quicksort method".

This method is simple and easy to implement. On the other hand, it has a certain weakness: during the computation, there are some redundant works wasting the processing resources. If we observe the windows, it is easy to find out that the neighboring windows shared a big part of same area, with only very few cells different between them. Comparing a window with its previous window (before moved), only few cells are removed and few cells are added, with most cells remaining unchanged. We show this situation in Figure 2.

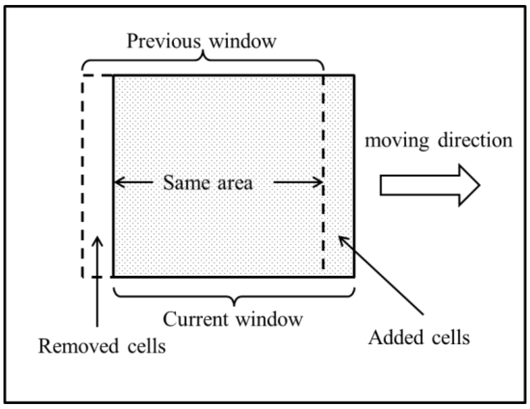

Figure 2 Comparison between adjacent windows

Quicksort method ignores this feature of adjacent windows and only computes every window separately. Actually, if we can somehow reuse the processed data in the previous window and compute the percentile incrementally through moving steps, it will become more efficient.

3.3 Proposed Method: Incremental Computation with Self-Balancing BST

Percentile is to compute the *n*-th smallest value among the current window. Here *n* is calculated from the given percentile *p*-th as introduced in Section 3.1 In order to obtain this quickly in each window, we propose to use the data structure of self-balancing binary search tree [14] to maintain the values of the current window in sorted orders. When moving to a new window, the tree contains all the values of the previous window, so only few insert and delete operations needed to modify it into a tree only containing the values of the new window. Then the current percentile can be computed quickly, as the old sorted values of the previous window in the tree reused, this is an incremental computation.

3.3.1 Self-balancing Binary Search Tree

It is necessary to first introduce this data structure, self-balancing binary search tree, or just self-balancing BST for short from now on in this paper.

BST (Binary Search Tree) is a simple data structure, also known as ordered/sorted binary tree. It is a node-based binary tree, each node with a comparable key and satisfies that the key in any node is larger than the keys from that node's left sub-tree's nodes and smaller than the keys from that node's right sub-tree's nodes. Insert and delete keys are both very fast. In addition, by recording the size of the sub-tree for each node, it is convenient to compute the *n*th (*n* can be any value) smallest key in the tree. This operation is called "selection" in BST and is exactly what we need for calculating the percentile function.

However, a simple BST is efficient only when it is balanced. Most operations, such as insertion, deletion and selection that we need, all take time directly proportional to the height of the tree. In our case, the tree would continuously adding and removing nodes, so it is very easy for the tree to turn into a bad unbalanced shape with big height, which leads to low efficiency.

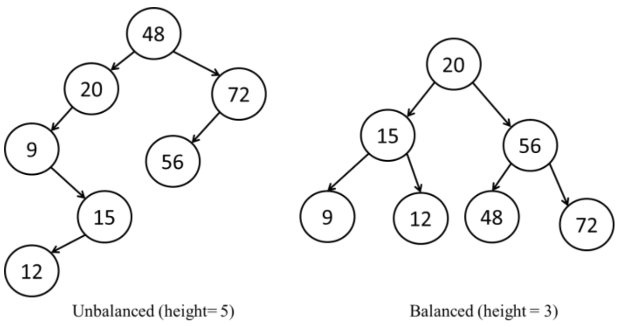

Figure 3 Example: two BSTs with same keys in different shapes

To solve this, self-balancing binary search tree is actually used, which can automatically keep its height small in the face of arbitrary insertions and deletions. This kind of BST adjusts its shape by some rotation operations under certain conditions, keeping it balanced or near-to-perfect balanced. In such tree, insertion, deletion and selection operations will all only cost O(log*N*) in amortized complexity. Here *N* is the total number of nodes in the tree.

3.3.2 Incremental Computation Process

By applying self-balancing binary search tree as the data structure to maintain the window values for incremental computation, we can take advantages of the close relationship between adjacent windows by reusing a lot of processed data from the previous window in the BST. Many redundant sorting works are reduced, that leads to high efficiency.

We here introduce the detailed process steps of the incremental computation for the percentile window aggregates. We treat the computation process into 2 stages. First stage is to generate basic windows from the first n-1 dimensions, then for every basic window, process the second stage, moving the window forward in the last dimension, calculate the percentile for each window incrementally.

Here a basic window is the first window in a moving round through the last dimension, so the position of their last dimension is always in the first, while the other n-1 dimensions' positions vary. As shown in the Figure 4, the grey areas are the basic windows.

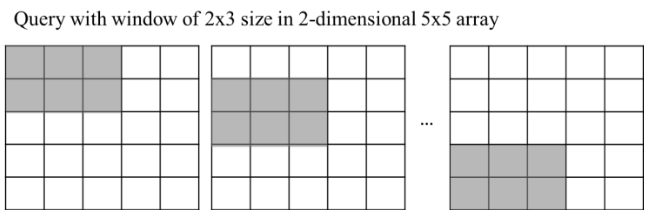

Figure 4 Basic windows of a window query in a 2-D array

Here we explain how the incremental computation method performs in detailed steps with an example small 2-Dimensional array in the following.

Step 1. Generate a basic window, initialize the self-balancing BST, and insert all the values of the basic window into the BST. The result of a selection operation which computes the *n*-th smallest key in the BST is the percentile value for the basic window.

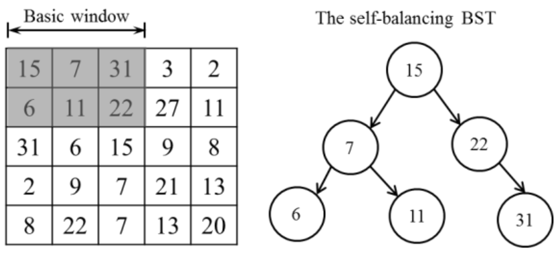

Figure 5 Basic window and Initializing the BST

Step 2. Moving the window forward in the last dimension, inserting new coming values into the self-balancing BST, while deleting the old values that no longer in the current window. After all the updating, the values inside the BST become exactly the values of the current window. This situation is shown if Figure 6. Again a selection to get the *n*-th smallest key in the tree would be the percentile result.

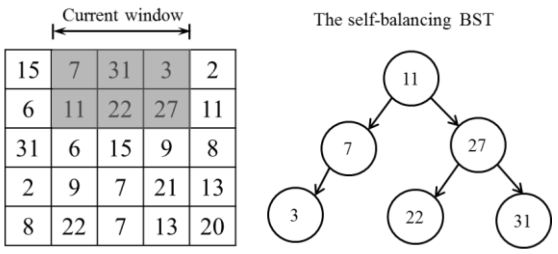

Figure 6 process a new window, update the BST

Step 3. Keep moving forwards, repeat step 2 to compute the aggregate percentile values of all the windows that derived from the basic window in step 1.

Step 4. Moving on to a new basic window, repeat the step 1~3, which is considered as a computation round. After finishing all the basic windows' computation rounds, the percentile window aggregate is done completely.

3.4 Parallel Processing

SciDB is a parallel array DBMS. Beside the special array data model, its parallel feature is another big advantage. By building a large cluster consists of multiple nodes, it can accelerate analytical tasks through its distributed system and run sub-tasks on each node at the same time.

Here we show our proposed method is processed in parallel with the architecture of distributed chunking in SciDB.

SciDB divides a large array into smaller chunks and distribute these chunks across a cluster to store. As build-in queries of SciDB, if a query can be processed by the unit of a chunk, and can collect sub-results of each chunk and figure out the merged result for the whole query, then this query can be processed in parallel.

When computing window aggregate query of percentile using our proposed incremental computing method in a cluster, every chunk can be assumed as an independent sub-array. Inside each chunk, a moving window is processed, as well as a self-balancing BST is maintained to compute the percentile incrementally. Figure 7 shows this situation.

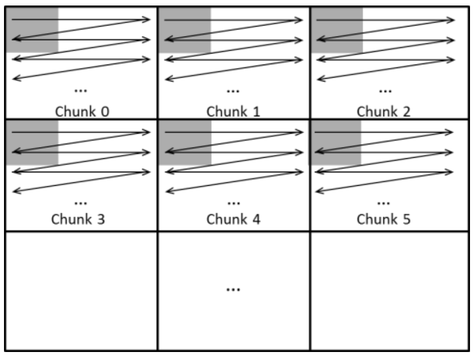

Figure 7 Incremental Computation in parallel

When gathering the sub-output into a final result array, there is no dependence between chunks. It simply constructs an array with the result of each chunk in its corresponding area. Then it obtains the final query result. Therefore, the proposed method is able to be computed in parallel, which is examined in Section experiment section 5 that describe the experimental evaluation.

4. Analysis

This Section analyzes time complexity of all the three methods introduced above to compute the window aggregate query with percentile operator.

For the preparation to describe the analysis, some parameters in window aggregates need to be defined so that the description is clear. For a n-dimensional array, its dimension sizes is defined as $D_1, D_2, \dots, D_n$. For a window aggregate query over such an array, the window size is specified in each dimension as $w_1, w_2, \dots, w_n$. By this definition, the total number of cells in this array is shown as $\prod_{i=1}^{n} D_i$ and total cell number of a normal window is shown as $\prod_{i=1}^{n} w_i$.

4.1 Quicksort Method

Quicksort method computes every window separately. Therefore, the time complexity is easy to compute.

First, let's consider the total number of windows. Each cell in an array has a corresponding window, thus the number of windows to be computed is same as the total cell number in an array, which is, in a n-dimensional array case, $\prod_{i=1}^{n} D_i$

Then let's consider the quicksort part. For each window, a quicksort needs to be processed, which sorts all the values in the window. Since there are $\prod_{i=1}^{n} w_i$ cells in a window, the time complexity for computing one window becomes as follows: $O(\prod_{i=1}^{n} w_i \, log\, (\prod_{i=1}^{n} w_i))$

From the analysis above, the total time complexity for quicksort method is:

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

4.2 Proposed Method

Because the incremental computation method is more complicated, consider a 2-dimensional array with dimension sizes $X \cdot Y$ first and set the window sizes of the query to be $a \cdot b$.

In this case, at total X basic windows exist. According to the compute process introduce in section 3.3, in each basic window's compute round, need to move on the second dimension (with size Y) to get new windows and process the incremental computing. That is Y moving steps at total. In other words, each basic window has Y derivative windows to be calculated.

When calculating each derivative window, values of a cells need to be inserted into the BST, while values of a cells need to be deleted from the BST. This is showed in Figure 8. After that, one selection operation needed to get the percentile result of the current window. Meanwhile, the self-balancing BST always maintains exactly all the cells of the current window inside it, so the size of the tree is always $a \cdot b$. Thus every single operation in the self-balancing BST, such as insertion, deletion and selection, always cost $\log ab$

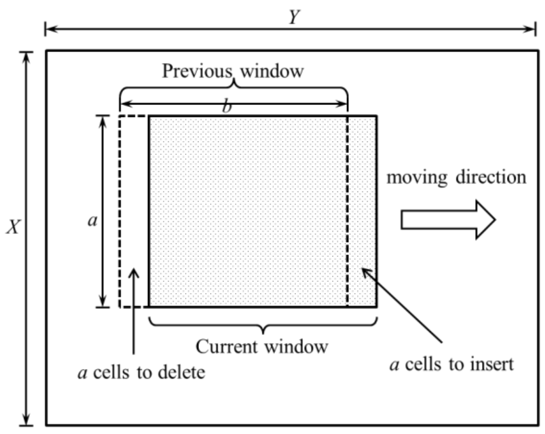

Figure 8 Details of window in one move step
2-D array with size $X \cdot Y$, window size as $a \cdot b$

From the analysis above, for each move step, adding up the cost for the insertions, deletions and selection, the time complexity is $O(a \cdot \log ab + a \cdot \log ab + \log ab)$, which can be simplified as $O(a \cdot \log ab)$

So the total time complexity of *2*-dimensional case is

$$O(XYa \cdot \log ab)$$

Also place the complexity of quicksort method here, with same definition of dimension and window sizes in 2-D case,

$$O(XYab \cdot \log ab)$$

It is obvious that our proposed method has a speedup by a factor of b in 2-Dimensional cases comparing with the quicksort method..

For a n-dimensional array, the analysis is similar.

First, let's consider the number of basic windows. Because basic windows' position are determined by the first n-1 dimensions, therefore the number of basic windows is $\prod_{i=1}^{n-1} D_i$. Then let's consider each basic window. In the computation round based on one basic window, the window is moved along the last dimension. Therefore D_n moving steps exist, thus D_n derivative windows need to be computed.

For each derivative window, the number of new cells to be inserted into the self-balancing BST is $\prod_{i=1}^{n-1} w_i$, same as the number of cells to be removed. After that, one selection executed to get the percentile result. When executing all these BST operations, the tree's size is the same as window's total size, that is $\prod_{i=1}^{n} w_i$. Therefore each single operation of the BST costs $\log(\prod_{i=1}^{n} w_i)$. So $\prod_{i=1}^{n-1} w_i$ times of insertions, $\prod_{i=1}^{n-1} w_i$ times of deletion and one time of selection, these are all the operations in the self-balancing BST need to be executed in one window. Summarize them up, the complexity for one window is

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

It can be simplified into

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

Finally, as analyzed above, $\prod_{i=1}^{n-1} D_i$ basic windows exist, for each basic window, D_n windows are to compute, that is, in total the time complexity for our proposed method is

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

Compare it with the quicksort method, whose complexity is

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

The incremental computation method gets a speedup factor of w_n according to the time complexity. This means that our proposed method would be about w_n times faster than quicksort method in theory.

5. Experiment

This chapter introduces the details of experiments, including the evaluation environment, the data used and performance comparison.

5.1 Environment

The experiments were executed over a SciDB cluster that consists of 9 nodes. Each node has the same environment and its configuration parameters are as follows:

Operating System	: CentOS 6.5
CPU	: Intel(R) Xeon(R) E5620 2.40GHz
Main Memory Size	: 24GB

Even for some special experiments on a single node, the configuration of that node is same as showed above.

5.2 Implementation

In order to evaluate the performance between the proposed method and the quicksort method, we implement these two methods into SciDB. Here is some information.

Language	: C++
Number of Lines	: 1300
SciDB version	: 13.12

The source codes of our implementation can be accessed on Github [20]. From source codes, only a plugin file can be built. It should be loaded into SciDB system to work correctly.

SciDB supports a convenient plugin-mechanism that allows users to implement their own defined operators. A user-defined-operator can be loaded into SciDB as a plugin. Once loaded, the plugin-operator can be executed exactly same as the built-in operators of SciDB. With plenty interfaces provided to access data from array, user can focus on implementing desired operator without considering the low-level's system issues.

5.3 Data

The data used for experiments is the JRA55 data [14]. The chosen data attributes and query parameter settings fit in real meteorological applications that require to computing percentile. The evaluated queries are required in real analysis by meteorologists.

For the experiment, we loaded 20 years' surface temperature data of JRA55 into the SciDB cluster. The size of one year's surface temperature data is about 4GB. When loading the data, we converted JRA55 files from GRIB2 format to CSV format since SciDB currently does not provide a mechanism to load GRIB2 format file whereas CSV is provided.

5.4 Performance Evaluation

5.4.1 Proposed Method vs. Quicksort Method

To compare the efficiency of these two methods, we designed two series of test cases, with different parameter settings.

The first series of evaluations are executed over an array containing the JRA55's surface temperature data of year 2012. The array is a 3-dimensional array of size 288×145×366, corresponding with longitude, latitude and time. The first two dimensions specify the location of the cell on earth, while the last dimension specifies which day's data the cell contains. In each cell, the main attribute is the surface temperature at 12 o'clock on that day.

The window aggregate query tested here is required by our cooperating meteorologists [16]. They wish to calculate percentiles over temporal windows, more specifically, to calculate percentiles of every 30 days' temperature data. Meanwhile, on the spatial aspect, they don't require overlap windows, so the computation is over single spatial cells. Figure 9 shows this.

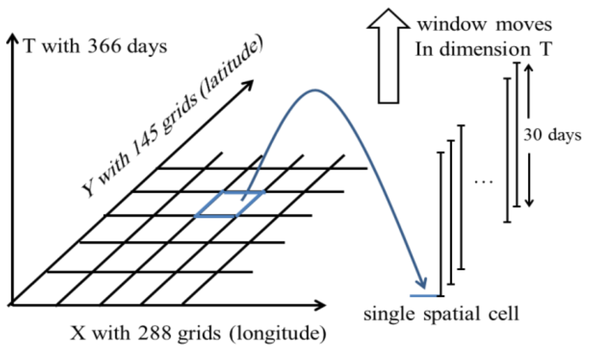

Figure 9 Window query in a real meteorological application, computing percentile in temporal window of 30 days.

Therefore, when designing the window parameters of the testing query, we set window size of the first two dimensions as 1x1, and only increase the window size in the temporal dimension. The tested attribute is the surface temperature at noon and the percentile percentage was set as 70%. The result is shown in Table 1 and Figure 10.

Table 1 Query Processing Time (in seconds) with JRA55 Data of Year 2012 (288x145x366)

Window Size	1x1x5	1x1x10	1x1x15	1x1x20	1x1x25	1x1x30
Balanced-BST	2.97 s	2.99 s	2.98 s	3.03 s	2.98 s	3.04 s
Quicksort	7.31 s	14.28 s	20.96 s	27.99 s	34.90 s	41.01 s
Speedup	2.46	4.78	7.03	9.25	11.71	13.49

Note: the balanced-BST represents the incremental competition method, while quicksort represents the naive sort method.

It shows improvements in running time of our proposed incremental computation method against the naive sort method. As the window size gets larger, the effect of improvement also gets larger. The last case with window size 1x1x30 is exactly the query needed in the meteorological analyze we mentioned. It gets a significant speedup by factor of 13.49.

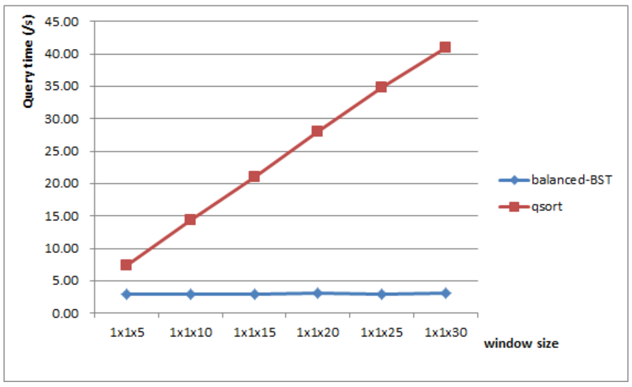

Figure 10 Query Processing Time with window Size

From the result above, a performance feature of the incremental computation method can be found. With all the other parameters fixed, no matter how w_n, the window size in last dimension varies, the processing time of the query almost remain the same. This feature is consistent with our time complexity analysis described in Section 4.2. As analyzed, the time complexity of our proposed method is

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

It is obvious that compared with other parameters, w_n makes very little contribution in this expression, only a factor inside log. This is the reason why the increasing of w_n almost has no effect to the query time in the evaluation above.

Here is the second series of test data. This time, we only change the window sizes in the first 2 dimensions (longitude and latitude) and remain the time dimension unchanged. When the window size becomes large, the quicksort method costs too much time, therefore we choose a smaller array than the one in the previous experiment, with only one month's data, the JRA55 data of 2012, January. This array contains temperature at 0, 6, 12, 18 o'clock in one day instead of only the data at 12 o'clock. Therefore the 3rd dimension of this array is 124 (31x4) instead of 31.

Table 2 Query Processing Time (in seconds) with JRA55 Data of Year 2012, January (288x145x124)

Window Size	1x1x5	2x2x5	3x3x5	4x4x5	5x5x5
Balanced-BST	1.26 s	4.44 s	9.83 s	18.38 s	28.75 s
Quicksort	3.03 s	11.32 s	26.39 s	47.00 s	72.24 s
Speedup	2.41	2.55	2.68	2.56	2.51

The result of experiment is shown in Table 2 and Figure 11. Again, the result shows improvements in efficiency of the proposed method for the percentile window aggregate query.

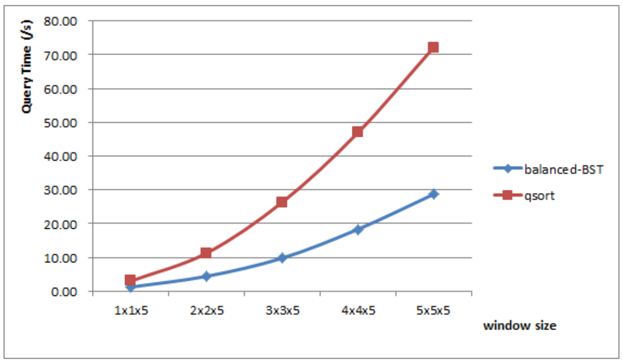

Figure 11 Query Processing Time with window Size

More can be found from the two test series above. We already understand from the time complexity analysis in section 4 that in theory, comparing with the quicksort method, our proposed method has a speedup factor of w_n, which in the above 3-dimensional cases, is w_3, that is the last window size parameter. The experiment results can prove this analysis. In this test series, as the w_3 remains the same, the speedup value of proposed method against the quicksort method also almost remains the same. Meanwhile, in the first test series, as the w_3 increases in linear, so does the speedup value. To show this more clearly, dividing speedup values with w_3 in every test case of both series, the results seem both to be a same constant.

This proved that our analysis of the speedup factor to be w_n is correct. It should be noted that of course, a constant factor exists here.

5.4.2 Cluster vs. Single Node

After the evaluation between two methods, how the incremental computation method performs in parallel should also be evaluated.

As introduced before, the cluster we used for testing consists of 9 nodes. To evaluate the parallel computing performance, a single node cluster was also built as a comparison. The same JRA55 dataset for evaluating was also loaded into this single node. The queries tested here is the same ones tested in Section 5.4.1, with window sizes of the query varying only in the first two dimensions.

Table 3 Parallel Test: Cluster (9 Nodes) vs. Single Node

Window Size	1x1x5	2x2x5	3x3x5	4x4x5	5x5x5
Cluster	1.26s	4.44s	9.83s	18.38s	28.75s
Single Node	5.90s	22.19s	48.13s	88.37s	136.53s
Speedup	4.68	5.00	4.90	4.81	4.75

The result of experiments is shown in Table 3. The experimental result shows that the cluster processes the same percentile window aggregate about 5 times faster than the single node server. It is acceptable since in a parallel computation across a database cluster, there would be other time cost besides query running time, such as communication cost between nodes. With the situation of analyzing huge amount of data by window aggregates, building a large cluster to calculate the query in parallel can be an efficient solution.

6. Conclusions

This paper proposes an efficient algorithm for percentile window aggregate query in array databases. By using the data structure of self-balancing binary search tree, lots of redundant work has been eliminated comparing to the naive quicksort method, leading a much better efficiency.

We implemented this method as well as the quicksort method in SciDB, which is an open-source array database system. Performance experiments were executed with real scientific data — the JRA55 dataset. The result showed excellent improvement. The improvement ratio was consistent with the time complexity analysis results. In summary, the incremental computation method has a speedup by a factor of w_n comparing with the naive sorting method. This w_n is one of the window query parameters, meaning the window size in the last dimension. We conclude that our proposed method succeeded to improve the efficiency of window percentile computation over array data.

Acknowledgement

Here we thank to Prof. Matsueda for his fruitful comments on a meteorological application with percentile. This work is partially supported by JSPS KAKENHI Grant Number 24240015C, JSPS KAKENHI Grant Number 25280043HA. It is also supported by JST CREST “System Software for Post Petascale Data Intensive Science” and JST CREST “Extreme Big Data (EBD) Next Generation Big Data Infrastructure Technologies Towards Yottabyte/Year”

Reference

1) Adam Seering, Philippe Cudre-Mauroux, Samuel M. Samuel etc. Efficient Versioning for Scientific Array Databases, ICDE, 2012.
2) Alex V. Ballegooij, Roberto Cornacchia, Arjen P. deVries, Martin Kersten. Distribution Rules for Array Database Queries. Database and Expert Systems Applications Lecture Notes in Computer Science Volume 3588, 2005, p 55-64
3) Emad Soroush, Magdalena Balazinska, and Daniel Wang. ArrayStore: A Storage Manager for Complex Parallel Array Processing. SIGMOD conference, 2011
4) Emad Soroush and Magdalena Balazinska. Time Travel in a Scientific Array Database. ICDE, 2013.
5) M.Stonebraker, J. Becla, D. DeWitt etc. Requirements for Science Data Bases and SciDB. CIDR Conference, Asilomar, CA, USA, 2009
6) M. Kersten, Y. Zhang, M. Ivanova. SciQL, A query language for science applications. EDBT/ICDT 2011 Workshop on Array Databases.
7) P. Cudre-Mauroux, H. Kimura, K.-T. Lim etc. A Demonstration of SciDB: A Science-Oriented DBMS. VLDB’09 Vol. 2, Num. 1, 1534-1537, Lyon, France, 2009
8) Paul G. Brown. Overview of SciDB, Large Scale Array Storage, Processing and Analysis. SIGMOD Conference, 2010
9) Tingjian Ge, Zdonik, S. Handling Uncertain Data in Array Database Systems. ICDE 2008. IEEE 24th International Conference.
10) SciDB Development team. SciDB User Guide version 12.10, 2012.
11) Ying Zhang, Martin Kersten, Milena Ivanova. SciQL: bridging the gap between science and relational DBMS. IDEAS 2011, p 124-133
12) Li Jiang, Hideyuki Kawashima; An Incremental Computation Scheme over Array Database. IPSJ SIG technical reports, Volume 2013-DBS-158, Issue 8, November, 2013.
13) A. Ebita, S. Kobayashi, Y. Ota etc. The Japanese 55-year Reanalysis “JRA-55”: An Interim Report, SOLA, 2011, Vol. 7, 149−152
14) JRA55 data online archive. http://gpvjma.ccs.hpcc.jp/~jra55/#
15) Daniel D. Sleator, Robert E. Tarjan; Self-Adjusting Binary Search Tree, Journal of the Association for Computing Machinery, Vol. 32, No.3, July 1985
16) M. Matsueda, T. Nakazawa. 2014: Early warning products for severe weather events derived from operational medium-range ensemble forecasts. Meteorol. Appl. doi:10.1002/met.1444.
17) http://en.wikipedia.org/wiki/Percentile
18) Matthew Moyers , Emad Soroush and etc. A Demonstration of Iterative Parallel Array Processing in Support of Telescope Image Analysis. VLDB, Volume. 6, Issue 12, August, 2013
19) Jacob VanderPlas, Emad Soroush and etc. Squeezing a Big Orange into Little Boxes: The AscotDB System for Parallel Processing of Data on a Sphere. IEEE Data Eng. Bull. 36(4): 11-20, 2013
20) Li Jiang. Source codes of the implementation. https://github.com/ljiangjl/Percentile-in-SciDB.git