Progressive Top-*k* Subarray Query Processing in Array Databases

Dalsu Choi
Korea University
Seoul, Korea
dalsuchoi@korea.ac.kr

Chang-Sup Park
Dongduk Women's University
Seoul, Korea
cspark@dongduk.ac.kr

Yon Dohn Chung*
Korea University
Seoul, Korea
ydchung@korea.ac.kr

ABSTRACT

Unprecedented amounts of multidimensional array data are currently being generated in many fields. These multidimensional array data naturally and efficiently fit into the array data model, and many array management systems based on the array data model have appeared. Accordingly, the requirement for data exploration methods for large multidimensional array data has also increased. In this paper, we propose a method for efficient top-k subarray query processing in array databases, which is one of the most important query types for exploring multidimensional data. First, we define novel top-k query models for array databases: overlap-allowing and disjoint top-k subarray queries. Second, we propose a suite of top-k subarray query processing methods, called PPTS and extend them to distributed processing. Finally, we present the results of extensive experiments using real datasets from an array database, which show that our proposed methods outperform existing naïve methods.

PVLDB Reference Format:
Dalsu Choi, Chang-Sup Park, and Yon Dohn Chung. Progressive Top-k Subarray Query Processing in Array Databases. *PVLDB*, 12(9): 989-1001, 2019.
DOI: https://doi.org/10.14778/3329772.3329776

1. INTRODUCTION

Unprecedented amounts of multidimensional array data have been generated in many scientific and industrial fields. The management of such data and the processing of scientific workloads in the relational data model can be highly inefficient [10, 24]. In the necessity of efficiently managing multidimensional array data, many array management systems based on the array data model have been proposed [6, 7, 20, 22, 28].

A top-k query outputs k records with the highest scores. These top-k records usually represent the most important answers among all possible answers in specific aspects, and therefore facilitate users understanding of the data. Top-k queries can also be used to obtain basic knowledge before the application of complex data analysis techniques. Top-k query processing in various fields has been extensively studied.

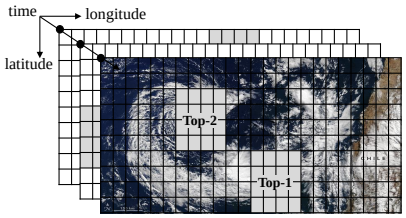


Figure 1: Two dimensional arrays with top-2 subarrays.

Array databases manage extremely large multidimensional array data, and therefore, it is difficult to understand data intuitively and extract important regions. To efficiently analyze and understand large array data, the support of top-k queries in array databases is required. Such queries have not been studied in spite of this need. The concept of top-k queries can be easily applied to multidimensional array databases as follows.

Example 1: Meteorologists monitor global cloudiness and its associated properties. Cloudiness monitoring plays an important role in predicting climate changes, because clouds affect the global energy balance. Frequently, the meteorologists objective is to find k fixed-sized regions sorted by a scoring function for the cloud top height that satisfy some selection conditions for the cloud top pressure in the observed cloudiness map. Figure 1 shows an example map [2] in which the shaded regions represent top-2 regions according to the given scoring function and selection conditions. The meteorologists extract interesting and important regions, study the change in k regions in time-series, and find unexpected changes.

Example 2: A group of researchers plans to create an Amazon deforestation detection system. The Amazon has a profound effect on ecological conditions worldwide. Therefore, monitoring of the deforestation in the Amazon is critical. The researchers extract the top-k fixed-size regions as scored by the average of the Normalized Difference Vegetation Index (NDVI). They also attempt to extract the top-k regions, where the regions are scored according to the average of the difference between neighboring arrays in the NDVI. This facilitates the detection of changes in forestation.

In the above example scenarios, it should be noted that the top-k results are an ordered sequence not of k cells, but

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 12, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3329772.3329776

of k subarrays. We call this novel type of query for multi-dimensional array databases the **top-k subarray query** in this paper. The top-k subarray query is useful for understanding and finding the meaning of array data.

To the best of our knowledge, no efficient methods exist for solving the top-k subarray query. In a naïve manner, all subarrays' scores in a given array are computed and the k subarrays with the highest scores are selected. However, the naïve method suffers two problems. First, the computation of all subarrays' scores in a large array is very time consuming. Second, until computing the scores of all subarrays, users cannot know the answers, so usability is severely decreased.

In this paper, we propose an efficient processing method for top-k subarray queries for array databases based on the following research directions. First, the top-k subarrays should be found without it being necessary to compute all the subarrays in the given array. Second, usability should be increased. Third, it is necessary to prevent memory overflow during the query processing in large array data. Fourth, distributed processing should be supported, because most array database systems basically support distributed environments because of the very large amount of array data that they manage.

We first define the top-k subarray query in array databases. Note that different subarrays may or may not overlap one another. Therefore, we define two different types of top-k subarray query: an **overlap-allowing** and a **disjoint top-k subarray query**. The overlap-allowing top-k subarray query reflects the fact that our objective is to find the subarrays with the highest scores, regardless of overlaps. In some cases, most k subarrays may overlap one another if a small region has extremely large attribute values. The results may be less meaningful, because the subarrays are located in almost the same position. Therefore, we define an additional top-k subarray query, the disjoint top-k subarray query. The disjoint top-k subarray query returns top-k subarrays disjointed from one another.

To solve these two types of top-k subarray queries, we preprocess the underlying array data. First, the array is partitioned to allow useful summary information to be extracted. We propose top-k subarray query processing methods called PPTS that utilize these partitions and prune search space to find top-k subarrays efficiently. It should be noted that a query is progressively solved. The progressive approach increases usability, because the user can perform data analysis using the partially returned subarrays while the query is finding the remaining answers. Then, we propose a distributed progressive top-k subarray query processing method. The support of distributed processing is a critical issue in large scale data analysis. Finally, we introduce optimization techniques for the proposed methods.

The rest of this paper is organized as follows. In the next section, we define two types of top-k subarray queries. Section 3 introduces a naïve method. We propose a partition-based progressive top-k subarray query processing method (PPTS) in Section 4. Section 5 introduces distributed processing for the proposed method and the naïve method. Section 6 presents optimization techniques. In Section 7, we describe the experimental evaluation of the methods. Section 8 introduces related work, and the final section concludes the paper.

2. PROBLEM DEFINITION

Definition 2.1 (**Array**). *Suppose that a dimension D_i is a set of consecutive integers in $[L_i, U_i]$, i.e., $D_i = \{x \in Z | L_i \leq x \leq U_i\}$ where the lower bound L_i and upper bound U_i are integers, and an attribute A_i is a set of real numbers. Given m dimensions $\{D_1, D_2, ..., D_m\}$ and n attributes $\{A_1, A_2, ..., A_n\}$, an m dimensional array A is defined by a function $D_1 \times D_2 \times ... \times D_m \mapsto (A_1, A_2, ..., A_n)$. A tuple $(d_1, d_2, ..., d_m)$ in $D_1 \times D_2 \times ... \times D_m$ is called a cell of the array.*

Definition 2.2 (**Subarray**). *Given an array A defined by $D_1 \times D_2 \times ... \times D_m \mapsto (A_1, A_2, ..., A_n)$, where $D_i = \{x \in Z | L_i \leq x \leq U_i\} (1 \leq i \leq m)$, a subarray SA of A is an array $Ds_1 \times Ds_2 \times ... \times Ds_m \mapsto (A_1, A_2, ..., A_n)$, where $Ds_i = \{x \in Z | L_i \leq Ls_i \leq Us_i \leq U_i\} (1 \leq i \leq m)$, and the attribute values of a cell $(d_1, d_2, ..., d_m)$ in SA are the same as those in A. $(Ls_1, Ls_2, ..., Ls_m)$ is the starting cell of a subarray and $(Us_1, Us_2, ..., Us_m)$ is the ending cell. A subarray is denoted by $SA(\text{starting cell}, |Ds_1| \times |Ds_2| \times ... \times |Ds_m|)$.* □

To rank subarrays and find the top-k ones, a **scoring function** SF is applied to a subarray, which aggregates the values of a selected attribute, called a measure attribute, in the cells of the subarray and produces a score value. In this study, we assume that a scoring function strictly increases with regard to a partial order on the sets of attribute values, i.e., given two sets of attribute values, $\{c_{11}, c_{12}, ..., c_{1k}\}$, and $\{c_{21}, c_{22}, ..., c_{2k}\}$, if $c_{1i} < c_{2i}$ for $\forall i \in \{1, 2, 3, ..., k\}$, $SF(\{c_{11}, c_{12}, ..., c_{1k}\}) < SF(\{c_{21}, c_{22}, ..., c_{2k}\})$. Example scoring functions are *Sum, Avg, Min, Max*, and *Median*. A **selection condition** SC is a Boolean function applied to a subset of attributes $\{A_1, A_2, ..., A_n\}$ of a subarray.

Definition 2.3 (**Top-k Subarray Query**). *Given an array A, a scoring function SF, selection conditions SC, a subarray size $|Ds_1| \times |Ds_2| \times ... \times |Ds_m|$, and an integer k, a **top-k subarray query** finds the k number of $|Ds_1| \times |Ds_2| \times ... \times |Ds_m|$-sized subarrays of A that have the highest scores obtained from the SF and satisfy SC.* □

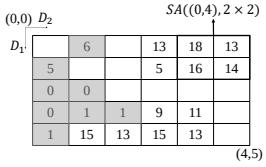
Note that the top-k subarrays can be disjoint or not. Therefore, we consider two types of top-k subarray queries: (1) the overlap-allowing top-k subarray query and (2) the disjoint top-k subarray query. An overlap-allowing top-k subarray query finds subarrays that satisfy Definition 2.3, regardless of overlaps among the resultant subarrays. k subarrays may or may not overlap one another. In some cases, an overlap-allowing top-k subarray query may ultimately include less meaningful k subarrays. That is, it is possible that most subarrays overlap one another if a small region in the given array contains many cells with extremely large values of the measure attribute. Thus, we define disjoint top-k subarray queries separately as follows.

Definition 2.4 (**Disjoint Top-k Subarray Query**). *Given an array A, a scoring function SF, selection conditions SC, a subarray size $|Ds_1| \times |Ds_2| \times ... \times |Ds_m|$, and an integer k, a **disjoint top-k subarray query** finds the k number of $|Ds_1| \times |Ds_2| \times ... \times |Ds_m|$-sized subarrays of A, $\{SA_1, SA_2, ..., SA_k\}$, that satisfy SC and the following conditions.*

- *SA_1 is the subarray having the highest score.*
- *$SA_i (2 \leq i \leq k)$ is the subarray that is disjoint from all $SA_j (1 \leq j < i)$ and has the highest score.* □

We call overlap-allowing top-k subarray queries **overlap-allowing queries** and disjoint top-k subarray queries **disjoint queries** in this paper.

Usually, a considerable amount of query processing time is required to find the top-k subarrays in a huge volume of multidimensional array data. To reduce the response time and increase usability, we propose a progressive method for top-k subarray query processing, called PPTS.



k	2
subarray size	2×2
scoring function	sum(temperature)
selection condition	avg(temperature) ≥ 12

Figure 2: Examples of a two dimensional array with a single attribute (e.g., temperature) and a top-k subarray query.

3. A NAÏVE METHOD

We first introduce a naïve method used to evaluate top-k subarray queries on array databases. This method computes all possible subarrays' scores and selects the top-k results, and therefore, is not progressive. The naïve method for overlap-allowing queries computes all the subarrays that satisfy the selection conditions with the given subarray size in the given array and stores only the k subarrays that have the highest scores during query processing. If a new subarray with a score higher than that of the k^{th} subarray in the candidate subarrays is found, it becomes a new candidate for the top-k answers, replacing the previous k^{th} one. The method allows overlaps among the top-k subarrays.

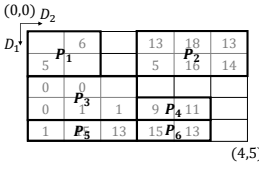
On the basis of the naïve overlap-allowing query processing method, we introduce a naïve method for disjoint queries. In disjoint queries, selected subarrays must be disjoint from one another. The method also computes all the subarray's scores in the given array; however, the difference between overlap-allowing and disjoint queries is that disjoint queries cannot determine whether a subarray is one of the top-k subarrays while computing subarrays, which means all the computed subarrays must be maintained. This is further discussed in Section 5.3. After checking all the subarrays, the method sorts them by scores and finds the disjoint top-k subarrays that satisfy Definition 2.4.

4. PPTS : PARTITION-BASED PROGRESSIVE TOP-K SUBARRAY QUERY PROCESSING

Irrespective of query type (overlap-allowing or disjoint), naïve methods compute all the possible subarrays in the array, which incurs an excessive computational cost. In Figure 2, it can easily be seen that, although it is unlikely that the subarrays including shaded cells belong to the top-k answers, they must be also computed. Furthermore, naïve methods cannot return top-k subarrays progressively. To overcome these problems, we propose a new top-k subarray query processing method which consists of two steps: partitioning the array and processing top-k subarray queries using the partitions.

4.1 Preprocessing - Partitioning over Array Data

To allow efficient processing of top-k subarray queries, we propose partitioning the given array data and computing the useful information about the partitions. The insight that suggested partitioning is that neighboring cells in array data tend to have similar attribute values. The partitioning results are used to prune the search space for finding top-k subarrays. Given a set of partitioning sizes $\{l_1, l_2, ..., l_m\}$ for the dimensions of an m dimensional array data, the entire array is divided into uniform subarrays called **partition**s. Then, each partition is shrunk as much as it can be while still containing all the cells in the partition.



	Starting cell	Ending cell	Max value	Count
P_2	(0,3)	(1,5)	18	6
P_5	(4,0)	(4,2)	15	3
P_6	(4,3)	(4,4)	15	2
P_4	(3,3)	(3,4)	11	2
...	...	...	...	...

Figure 3: Partitions when $l_1 = 2$ and $l_2 = 3$.

The partitioning algorithm is as follows. First, the given array is partitioned uniformly with partition size parameters $\{l_1, l_2, ..., l_m\}$, the partitions are shrunk, and their maximum values and cell counts of the measure attribute are calculated. Second, the partitions are sorted in descending order of maximum values. Finally, information about the partitions, including starting cells, ending cells, maximum values, and cell counts, are stored on disk in a partition table. Figure 3 shows the results of partitioning with partition size parameters $\{2, 3\}$ in the given example array.

The time complexity of partition computation is $O(n + NlogN)$, where n is the number of cells in the given array, and N is the number of partitions produced, since it is necessary to scan the entire array data and sort the resulting partitions. If the measure attribute is changed to the other attribute, partitioning should be performed for the new measure attribute.

4.2 Basic Partition-based Progressive Top-k Subarray Query Processing

We first introduce the maximal virtual subarray (MVS) and upper bound score (UBS).

Definition 4.1 (**Maximal Virtual Subarray, MVS**). *Given a partition p_i and a subarray's size $|Ds_1| \times ... \times |Ds_m|$, the **maximal virtual subarray** of p_i is a $|Ds_1| \times ... \times |Ds_m|$-sized subarray where all the cells have the maximum value of the measure attribute in p_i.*

Definition 4.2 (**Upper Bound Score, UBS**). *Given a partition p_i and a scoring function SF, the **upper bound score** of p_i is $SF(MVS(p_i))$.*

Lemma 4.1. *Suppose that there are a scoring function SF and a sequence of partitions $P = \{p_1, p_2, ..., p_n\}$ on an array in descending order of their maximum values. If a subarray SA overlaps with $p_i (1 \le i \le n)$ but does not overlap with any partitions in $P' = \{p_1, p_2, ..., p_{i-1}\}$, $SF(SA) \le UBS(p_i)$.*

Proof. When $i = 1$, $MVS(p_1)$ consists of the maximum value in the entire array, and thus, $SF(SA) \le UBS(p_1)$. For $1 < i \le n$, assume that $SF(SA) > UBS(p_i)$. There exists at least one cell c in SA that has a value larger than p_i's maximum value m. Since all the cell values in p_i are smaller than or equal to m, c does not belong to p_i. Then, c must belong to another partition p', the maximum value of which is obviously larger than or equal to the value of c. This means that p' belongs to P' and SA overlaps with a partition in P', which is a contradiction of the assumption. □

Corollary 4.1. *Given a scoring function SF and a set of partitions P on an array, if a subarray SA overlaps with a subset of partitions $P_s \subseteq P$ and the partition $p_m \in P_s$ has the highest maximum value among partitions in P_s, then $SF(SA) \leq UBS(p_m)$.*

PROOF. SA overlaps with p_m but does not overlap with any partition which has a higher maximum value than p_m. Therefore, $SF(SA) \leq UBS(p_m)$ by Lemma 4.1. □

If we consider selection conditions SC when getting UBS, UBS could be lowered; however, it is impossible for UBS to grow larger, which means that UBS still works.

Based on the previous definitions and lemma, the top-k subarrays can safely be returned progressively without checking all the subarrays in the given array. We describe partition-based progressive disjoint top-k subarray query processing in Algorithm 1. To find the i^{th} answer ($1 \leq i \leq k$) in the given array, the algorithm selects partitions serially in the sorted order and calculates the scores of all the unchecked subarrays related to each partition. Note that partitions are sorted in descending order of their max values. Thus, the algorithm searches the partitions that have higher maximum values earlier, which means that subarrays with higher scores are considered first. *currentTop* in Algorithm 1 keeps a subarray with its score that is the most promising candidate of the i^{th} answer. The algorithm checks the first unchecked partition in P and calls the procedure *getScores* on the partition. For each subarray in the set *overlappingSA* of unchecked subarrays overlapping with the partition, if it satisfies the selection conditions, the algorithm calculates the score (Lines 26-27). To find *overlappingSA*, the algorithm utilizes a Boolean array to determine whether each subarray has been already checked (Line 24). If the subarray's score is smaller than or equal to that of *currentTop*, the algorithm inserts the subarray into *Candidates* (Line 34); otherwise, if it is disjoint from all subarrays in $TopK$, *currentTop* and *Candidates* are updated with the subarray (Lines 28-32). *Candidates* is used to keep a set of subarrays that can be an answer to the query following the i^{th} answer.

After processing the partition, the algorithm goes to the next partition and checks the answer-returning condition (Lines 6-8). If *currentTop*'s score is higher than the partition's upper bound score, *currentTop* can safely be returned as the i^{th} answer (Line 14). Because the partitions are checked in descending order of maximum values, the UBS of a partition is the possible maximal score that an unchecked subarray overlapping with the partition can have. Therefore, if *currentTop* satisfies the answer-returning condition, there exist no unchecked subarrays in the remaining partitions that have higher scores than *currentTop*. *currentTop* should be inserted into $TopK$ and is used to filter candidate subarrays that overlap with the existing answers (Line 15). After returning the i^{th} answer, the algorithm removes unnecessary subarrays from *Candidates* and selects the subarray with the highest score in *Candidates* as new *currentTop* (Lines 16-21).

Although the overlap-allowing query processing algorithm is similar to Algorithm 1 for disjoint queries, there are two major differences. First, after returning the i^{th} answer, it does not consider overlaps and selects the subarray with the highest score in *Candidates* as a new *currentTop*. Second, after calculating a subarray's score (Line 27), an overlap-allowing query does not consider the disjoint condition. If a subarray satisfies the selection conditions, it is inserted into *Candidates*. If the score of subarray is higher than that of *currentTop*, it becomes new *currentTop*, without the overlaps being considered.

Algorithm 1: PPTS for a disjoint query

```
   Input : k, subarray size SS, scoring function SF,
           selection conditions SC, sorted partitions P
           on an array A
   Output: disjoint top-k subarrays
1  currentTop ← (null, −∞);
   // a set of top-k subarrays selected thus far
2  TopK ← ∅;
3  priority queue Candidates ← ∅;
4  for i=1 to k do
5  |  foreach unchecked partition p ∈ P do
   |  |  // Answer-returning condition
6  |  |  if currentTop.score > UBS(p) then
7  |  |  |  break;
8  |  |  end
   |  |  // Compute the scores of all unchecked
   |  |     subarrays overlapping with p
9  |  |  getScores(p,currentTop,TopK);
10 |  end
11 |  if currentTop=(null,-∞) then
12 |  |  quit;
13 |  end
14 |  output currentTop as the i^th answer;
15 |  TopK ← TopK ∪ {currentTop.SA};
16 |  From Candidates, remove subarrays that overlap
   |   with at least one subarray in TopK;
17 |  if Candidates = ∅ then
18 |  |  currentTop ← (null, −∞);
19 |  else
20 |  |  currentTop ← Candidates.pop();
21 |  end
22 end

23 Procedure getScores(p,currentTop,TopK):
24 |  overlappingSA ← a set of unchecked subarrays of
   |   size SS and overlapping with partition p;
25 |  foreach SA ∈ overlappingSA do
26 |  |  if SA satisfies SC then
27 |  |  |  score ← SF(SA);
28 |  |  |  if score > currentTop.score then
29 |  |  |  |  if SA is disjoint from all subarrays in TopK
   |  |  |  |   then
30 |  |  |  |  |  currentTop ← (SA, score);
31 |  |  |  |  |  Candidates ← Candidates∪{currentTop};
32 |  |  |  |  end
33 |  |  |  else
34 |  |  |  |  Candidates ← Candidates ∪ {(SA, score)};
35 |  |  |  end
36 |  |  end
37 |  end
```

We introduce an example of PPTS based on the example array and query in Figure 2 and the partitions in Figure 3. Assume that the query finds disjoint top-2 subarrays. The query visits the first partition P_2, the starting cell of which is (0,3). After all the subarrays overlapping with the partition are computed, *currentTop* is {(0,4),61}. The query reads the next partition P_5; its UBS is 60, and therefore,

currentTop satisfies the answer-returning condition. After the third partition P_6 is visited, *currentTop* becomes {(3,3),48}. Its score is higher than the UBS of the fourth partition P_4, 44. The query returns {(3,3),48} as the second answer without calculating the remaining subarrays.

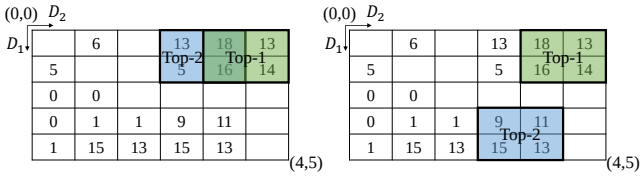


Figure 4: Results of the overlap-allowing query (left) and the disjoint query (right).

Theorem 4.1. *Given a scoring function SF, a sequence of partitions $P = \{p_1, p_2, ..., p_n\}$ on an array in descending order of their maximum values, and a top-k subarray query, assume that a set of partitions $P' = \{p_1, p_2, ..., p_{i-1}\}(1 < i \leq n)$ has been processed by the proposed algorithms. Let ANS be a subarray having the highest score among all the checked subarrays that are candidates for the next answer. If the score of ANS is larger than the UBS of the partition p_i, i.e., $SF(ANS) > UBS(p_i)$, there exist no unchecked subarrays having scores larger than ANS's score.*

Proof. Assume that there exists an unchecked subarray SA, the score of which is larger than that of ANS, i.e., $SF(SA) > SF(ANS)$. Since all subarrays overlapping with partitions in P' have already been computed, SA does not overlap with any partition in P', whereas it overlaps with at least one partition in $P - P'$. Without loss of generality, let $p_j (i \leq j \leq n)$ be a partition in $P - P'$ that overlaps with SA and has the largest maximum value. This means that SA does not overlap with $\{p_1, p_2, ..., p_{j-1}\}$, whereas it overlaps with p_j. By Lemma 4.1, $SF(SA) \leq UBS(p_j)$ and, by the order of partitions, $UBS(p_j) \leq UBS(p_i)$. Thus, if $SF(ANS) > UBS(p_i)$, we have $SF(SA) < SF(ANS)$, which is a contradiction of the assumption. □

5. DISTRIBUTED PROCESSING

5.1 Chunking Strategy of Array Databases

Before proposing distributed processing for top-k subarray query processing, we introduce the notion of chunking in array databases. If an array is too large to be stored in one node, array databases must spread the array across multiple nodes. For this purpose, array databases divide an array into multiple chunks [6, 7, 20, 22, 28]. Chunking is a basic principle for supporting distributed processing in array databases. A chunk becomes the unit of I/O and network transfer. SciDB [7], the de facto standard array database, divides an array into regular chunks, which means that each chunk has the same dimension sizes. SciDB requires users to input each dimension's size for chunks. ArrayStore [22] provides regular chunking, irregular chunking, and two-level chunking in regular or irregular fashions. Irregular chunking results in each chunk having different dimension sizes; the chunks satisfy different conditions, such as that the number of non-empty cells must be the same. We used the regular chunking strategy in this study. Our proposed methods can be easily extended for application to irregular chunking.

We assume column store array databases, which means that they store each attribute's values as separate chunks.

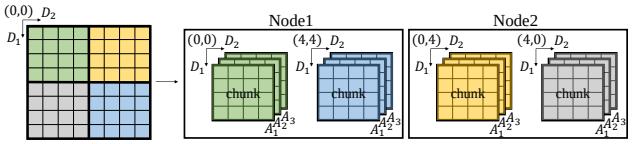


Figure 5: Regular chunking and distribution over two nodes.

That is, if an array has n attributes $\{A_1, ..., A_n\}$, the attribute values of a cell are stored on separate n chunks, all of which are stored in one node.

Figure 5 shows regular chunking in an array with two dimensions $\{D_1, D_2\}$ and three attributes $\{A_1, A_2, A_3\}$. The chunk sizes for D_1 and D_2 are 4. The chunks are distributed over two nodes and each node has six chunks.

5.2 Computing Subarrays on Chunk Boundaries

When the chunks of an array are distributed over multiple nodes, all the **boundary subarray**s that are on chunk boundaries cannot be calculated within a single node. In Figure 5, for example, Node 1 cannot calculate the 2×2 subarrays' score starting at (3,3), because it does not have cells at (3,4) and (4,3). To compute boundary subarrays, chunks have additional cells from other chunks according to the given subarray size. Given a subarray size $|Ds_1| \times ... \times |Ds_m|$, we need chunk overlaps of size $|Ds_i| - 1$ in each dimension $D_i (1 \leq i \leq m)$ to compute all the boundary subarrays.

Definition 5.1 (Chunk Overlap). *Given a set of overlap sizes $X = \{x_1, ..., x_m\}$, an m dimensional array, the starting cell of which is $\{L_1, ..., L_m\}$ and the ending cell $\{U_1, ..., U_m\}$, and a chunk, the starting cell of which is $\{Lc_1, ..., Lc_m\}$ and the ending cell $\{Uc_1, ..., Uc_m\}$, with a set of cells O, the chunk is extended by X with the starting cell $(max(Lc_1 - x_1, L_1), ..., max(Lc_m - x_m, L_m))$ and the ending cell $(min(Uc_1 + x_1, U_1), ..., min(Uc_m + x_m, U_m))$. When the extended chunk has a set of cells E, the chunk overlap means $E - O$.* □

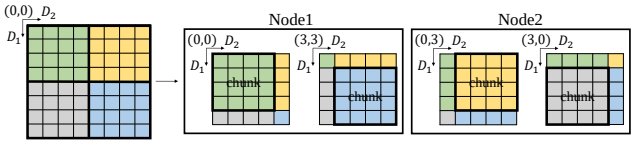


Figure 6: Chunks with $\{1, 1\}$-sized overlaps for an attribute when a subarray size is 2×2.

If we compute 2×2 subarrays, each chunk needs $\{1, 1\}$-sized overlaps to allow computation of boundary subarrays, as shown in Figure 6. $\{1, 1\}$-sized overlaps allow each chunk to compute all boundary subarrays; larger overlaps also do, but are unnecessary. SciDB [7] adopts the chunk overlap mechanism when processing window aggregates. Note that each chunk cannot know the exact sizes of the overlaps it needs before a top-k subarray query is given at runtime, although the overlaps of each chunk can be pre-defined.

5.3 Distributed Processing of a Naïve Method

On the basis of the concepts of chunking and chunk overlaps, we introduce a naïve method that can be applied in distributed array databases. To process an overlap-allowing query, the query determines the sizes of chunk overlaps based on the given subarray size and transfers cells among

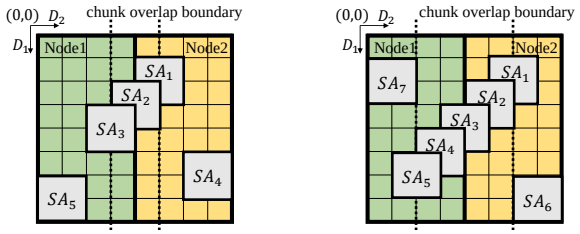
nodes to attain the chunk overlaps. Then, each node computes the scores of all the subarrays that satisfy the selection conditions, keeping the k subarrays with the highest scores. The k subarrays can be overlapped with one another. We call the k subarrays after computing all possible subarrays in each node **local top-k subarrays**. Each node sends its local top-k subarrays to the master node. The algorithm sorts the subarrays and selects the k subarrays with the highest scores as the **global top-k subarrays**.

Distributed processing of the naïve method for a disjoint query is almost the same as that for an overlap-allowing query. The major difference is that, in a disjoint query each slave node must keep all the computed subarrays in descending order of their scores rather than the local top-k subarrays. The master node receives the subarrays from all the slave nodes until it can decide global top-k subarrays. It should be noted that the distributed processing of a disjoint query is a holistic problem [13]; that is, the aggregation of the local top-k results from all the slave nodes is not sufficient to allow the master node to determine the global top-k results.

Lemma 5.1. *Given a disjoint top-k subarray query $F(k, A)$ on an array A with a scoring function SF and multiple chunks with chunk overlaps according to the given subarray size over $N(\geq 2)$ nodes, let (1) A_j be a subset of A in Node $j(1 \leq j \leq N)$ such that $A = \cup_{1 \leq j \leq N} A_j$, (2) $Top(F(k, A))$ be the set of disjoint top-k subarrays from $F(k, A)$ (if there do not exist disjoint k subarrays in A, $|Top(F(k, A))|$ could be smaller than k), and (3) M be the maximum integer in $\{m | 1 \leq j \leq N, m < m', |Top(F(m', A_j))| = m\}$. Then, it does not always hold that $Top(F(k, A)) \subseteq \cup_{1 \leq j \leq N} Top(F(M, A_j))$.*

Proof. Assume that $Top(F(k, A)) \subseteq \cup_{1 \leq j \leq N} Top(F(M, A_j))$ always holds. There could be an example that a subarray $SA_a \in Top(F(M, A_a))$ and a subarray $SA_b \in Top(F(M, A_b))(a \neq b)$ overlap each other, and Node a cannot compute SA_b. It is obvious that at least one of SA_a and SA_b, SA_{wrong}, cannot be a member of global top-k subarrays. There could be a real global answer SA_{real} that was not selected as a member of $Top(F(M, A_a))$ or $Top(F(M, A_b))$, because (1) $SF(SA_{real}) < SF(SA_{wrong})$, and (2) SA_{real} overlaps with SA_{wrong}. It cannot be guaranteed that this case does not occur, because (1) it cannot be guaranteed that any SA_a and SA_b are always disjoint from each other and (2) when SA_a and SA_b overlap with each other, Node a or b may not know that SA_b or SA_a, respectively, exist. Regardless of the size of M, $\cup_{1 \leq j \leq N} Top(F(M, A_j))$ cannot include SA_{real} in this case. The counter-example makes the assumption false, which means a disjoint top-k subarray query is a holistic problem. □

Figure 7 shows example scenarios, where each node contains only one chunk and the subarrays are numbered in descending order of their scores. In Figure 7(a), the sizes of chunk overlaps are $\{1, 1\}$, because the size of the subarray is 2×2. Disjoint top-2 subarrays, SA_1 and SA_4, from Node 2 and disjoint top-2 subarrays, SA_2 and SA_5, from Node 1 do not include one of the real answers, SA_3. Although chunks may have larger overlaps, the problem remains, as shown in Figure 7(b), where each node keeps only the top-3 disjoint subarrays. Therefore, the naïve method for disjoint top-k queries is not practical when an array is so large that nodes cannot keep all the computed subarrays in the main memory.



(a) $k = 2$ (b) $k = 3$

Figure 7: The problem of the naïve method for disjoint top-k subarray queries.

5.4 Distributed Partition-based Progressive Top-k Subarray Query Processing

Before top-k subarray queries are processed, the array is partitioned as a preprocessing step. In distributed array databases, chunks are partitioned independently of one another for the measure attribute. The results are stored in the same node with the chunks to process future top-k subarray queries. Note that partitioning does not consider chunk overlaps, because the future subarray size cannot be known.

We introduce PPTS in distributed environments in Algorithms 2 and 3. We assume that there are N slave nodes where the chunks of array data are stored. The master node runs Algorithm 2 and each slave node executes Algorithm 3. Partitions for each chunk in a slave node are sorted in descending order of their maximum values.

Given a top-k subarray query, the slave nodes calculate the size of chunk overlaps based on the given subarray size and transfer cells to one another to form the chunk overlaps if chunks do not have enough overlaps. To determine the $i^{th}(1 \leq i \leq k)$ answer, the master node requests new local answers from the slave nodes in S', which initially includes all slave nodes. Each slave node that receives the request processes its own chunks in descending order of the UBS of the next unchecked partitions in the chunks. This allows each node to visit chunks having partitions with higher UBS values first, which means that the probability that it will find a local answer with a higher score earlier is greater. If the current local answer's score is higher than a chunk's UBS (Line 14, Algorithm 3), the algorithm does not need to explore the chunk and thus it can avoid unnecessary chunk I/O. If this condition is not satisfied, but the chunk satisfies Line 19 (Algorithm 3) after checking one or more partitions, the query can proceed to the next chunk. When checking a partition, all the unchecked subarrays overlapping with the partition of the current chunk, including boundary subarrays, are computed using the procedure defined in Algorithm 1 (Line 22, Algorithm 3). If the current local answer satisfies Line 14 or 19 of Algorithm 3 for all chunks in the slave node, it is determined as the local answer of the slave node and sent to the master node (Line 25, Algorithm 3). When the master node has received local answers from all the slave nodes in S' (Lines 4-6, Algorithm 2), it selects the best one i.e., that with the highest score among the local answers of all slave nodes in S. After returning it as the global i^{th} answer, the master node sends the global answer to all the slave nodes in S (Lines 7-10, Algorithm 2). Each slave node receives the global answer (Line 27, Algorithm 3). If the global answer is not sent to all slave nodes right after it is decided as the i^{th} answer, slave nodes may select wrong local answers overlapping with the global answer

when deciding the subsequent global answers. The master node identifies a set S' of the slave nodes, the current local answers of which overlap with the global answer (Line 11, Algorithm 2) and requests the subsequent local answers from the slave nodes in S'.

Algorithm 2: (Master node)
Distributed processing of PPTS

```
   Input  : k
   Output : disjoint top-k subarrays
 1 Let S be a set of N slave nodes;
 2 S' ← S;
 3 for i=1 to k do
 4 |  foreach slave ∈ S' do
 5 |  |  Request and receive the local answer in slave;
 6 |  end
 7 |  globalAnswer_i ← the best among all local answers of
   |    the nodes in S;
 8 |  output globalAnswer_i as the i^th answer;
 9 |  if i < k then
10 |  |  Send globalAnswer_i.SA to all slave nodes in S;
11 |  |  S' ← a set of slave nodes in S, the current local
   |  |    answers of which overlap with globalAnswer_i.SA;
12 |  end
13 end
14 Notify all the slave nodes in S of termination;
```

Note that all slave nodes share the global answers immediately after they are determined by the master node, and therefore, the problem in the distributed version of the naïve method for disjoint queries can be avoided. In Figure 7(a), Node 1 selects SA_2 as the local answer, and Node 2 performs SA_1 using Algorithm 3. The master node receives Nodes 1 and 2's local answers and determines that SA_1 is the global top-ranked answer, because SA_1's score is higher than SA_2's. The master node notifies Nodes 1 and 2 of the global answer. Node 1 can select SA_3 as the subsequent local answer, because SA_2 is removed from $Candidates$. Node 2 selects SA_4 as the next local answer. After the local answers are sent/received, SA_3 is selected as the subsequent global answer.

When subarrays overlapping with a partition that was made without considering chunk overlaps are computed, boundary subarrays are also computed if they exist. Although partitions do not contain cells in chunk overlaps, PPTS guarantees that the global answers that lie on the boundaries of chunks are checked.

Lemma 5.2. *Given a sequence of partitions and a top-k subarray query on an array that has chunks distributed among $N(\geq 2)$ nodes, assume that there exists a boundary subarray SA which is the $i^{th}(1 \leq i \leq k)$ highest answer to the query. SA is selected as the local answer by one of the slave nodes SN that include a chunk overlapping with SA when the proposed progressive algorithm finds the i^{th} answer to the query.*

Proof. Assume that SA is not selected as a local answer by any node in SN. Two cases are possible: (1) SA is not checked by any node, and (2) although SA is checked by some node, it is not selected as the local answer of the node. First, Case (1) means that in Node $j \in SN$, a partition p_j that has the highest UBS among the partitions overlapping with SA is not processed. By Corollary 4.1, this means that $SF(SA) \leq UBS(p_j)$. When searching the i^{th} answer to the query, Algorithm 3 in Node j finds a local answer $localAnswer$ as a candidate for the global i^{th} answer. As p_j is not processed by Node j, $UBS(p_j) < SF(localAnswer)$. Therefore, we have $SF(SA) < SF(localAnswer)$. Since the master node running Algorithm 2 selects one of the local answers from all the slave nodes as the global i^{th} answer, it is a contradiction of the assumption that SA is the i^{th} answer to the query. For Case (2), we consider the overlap-allowing query and the disjoint query separately. There exists a subarray SA_{wrong} that has the higher score than SA, satisfies the selection conditions, and is selected as the local answer of a slave node in SN. In the overlap-allowing query, because the subarray with the highest score among local answers is selected as the global answer without considering overlaps, it is a contradiction of the assumption of this lemma. In the

Algorithm 3: (Slave node)
Distributed processing of PPTS

```
   Input  : chunks, k, subarray size SS, scoring function
            SF, selection conditions SC, sorted
            partitions on each chunk
 1 localAnswer ← (null, −∞);
 2 globalAnswers ← ∅;
 3 priority queue Candidates ← ∅;
 4 Determine the sizes of chunk overlaps based on SS, and
     transfer cells among nodes to form the chunk overlaps
     if chunks do not have enough overlaps;
 5 while True do
 6 |  Wait until a message m from the master node arrives;
 7 |  switch m do
 8 |  |  case m requests the subsequent local answer do
 9 |  |  |  if Candidates ≠ ∅ then
10 |  |  |  |  localAnswer ← Candidates.pop();
11 |  |  |  end
12 |  |  |  Sort the chunks in the node in descending order
   |  |  |    of UBS(nextPartition);
13 |  |  |  foreach chunk c in the current node do
14 |  |  |  |  if localAnswer.score > UBS(c.nextPartition)
   |  |  |  |    then
15 |  |  |  |  |  continue;
16 |  |  |  |  end
17 |  |  |  |  Read chunk c;
18 |  |  |  |  for p=c.nextPartition to c.lastPartition do
19 |  |  |  |  |  if localAnswer.score > UBS(p) then
20 |  |  |  |  |  |  break;
21 |  |  |  |  |  end
22 |  |  |  |  |  getScores(p,localAnswer,globalAnswers);
23 |  |  |  |  end
24 |  |  |  end
25 |  |  |  Send localAnswer to the master node;
26 |  |  end
27 |  |  case m delivers globalAnswer_i.SA do
28 |  |  |  globalAnswers ←
   |  |  |    globalAnswers ∪ {globalAnswer_i.SA};
29 |  |  |  Remove subarrays from Candidates that overlap
   |  |  |    with at least one subarray in globalAnswers;
30 |  |  end
31 |  |  case m notifies termination do
32 |  |  |  quit;
33 |  |  end
34 |  end
35 end
```

disjoint query, the existence of SA_{wrong} means that the node in SN does not have at least one of the previously selected global answers, $globalAnswers$. Note that after a local answer is selected as one of the global answers by the master node, the answer is shared among all slave nodes immediately. Algorithm 2 and 3 remove candidate subarrays that overlap with $globalAnswers$ before finding the i^{th} answer. Also, the procedure in Algorithm 1 does not allow subarrays overlapping with $globalAnswers$ to be a candidate. It is impossible for SA_{wrong} to be disjoint with $globalAnswers$, which means that Case (2) cannot be happened. Therefore, SA is safely selected as the local answer of a node in SN. □

Some boundary subarrays may be evaluated redundantly by several chunks. However, the case rarely happened in our experiments because (1) the number of boundary subarrays was small compared to the total number of subarrays and (2) there were few cases that all of the attribute values in a boundary subarray were large enough for all adjacent chunks to check it.

6. OPTIMIZATIONS

6.1 Incremental Computation

We consider an optimization technique to calculate the scores of the subarrays overlapping with a partition efficiently. If the scoring function is distributive or algebraic, such as *sum* or *average*, we can divide a subarray into several units by the most minor dimension Ds_m and compute the scores of consecutive subarrays, avoiding repetitive calculation of the same units. We explain the optimization technique using the example in Figure 2. When we check $SA((0,3), 2 \times 2)$, the subarray can be sliced into two units $\{U_1, U_2\}$ starting at $(0,3)$ and $(0,4)$. The scores of units are computed and summed to obtain the subarray's score. To compute the score of next subarray $SA((0,4), 2 \times 2)$, we do not need to recalculate the score of U_2. We need only the score of the unit U_3 starting at $(0,5)$. By subtracting U_1's score and adding U_3's score from the first subarray's score, we can obtain the second subarray's score. This optimization technique is similar to that presented in [15], but in this study we applied incremental computation not to the entire array but to the subarrays overlapping with a partition.

6.2 Upper Bound Score Optimization for Sparse Arrays

As mentioned in Section 4.2, the UBS of a partition p is derived from the MVS in which all the cells have the maximum value of the measure attribute in p. However, query processing using the UBS could be highly inefficient in the case of sparse arrays, because the UBS may be considerably higher than the scores of actual subarrays overlapping with the partition. Therefore, we devise an optimization technique applied on the UBS for sparse arrays based on maximal density estimation at query runtime. The optimization can be applied when a scoring function SF satisfies the condition: given a set of attribute values $C = \{c_1, c_2, ..., c_m\}$, $SF(C \setminus \{c_i\}) < SF(C) \quad (1 \le i \le m)$. Given a top-$k$ subarray query with a subarray size SS, it estimates the maximal density of each chunk that an SS-sized subarray can have. At query runtime, the UBS is calculated from the MVS that have the estimated density.

The primary concept of the maximal density estimation is that the preprocessing results, partitions, are utilized. Note that partitions have the number of non-empty cells, and therefore, we can infer the specific subarray's maximum cell counts.

Algorithm 4: Maximal density estimation

Settings: N slave nodes
Input : partitions, subarray size $|Ds_1| \times ... \times |Ds_m|$, partition size $\{l_1, ..., l_m\}$
Output : maximal density estimates for each chunk

1 $start \leftarrow \{0, ...\}$, $end \leftarrow \{0, ...\}$;
2 $newSubarray \leftarrow \{0, ...\}$, $newOverlap \leftarrow \{0, ...\}$;
3 **for** *i=1 to m* **do**
4 $\quad start[i] \leftarrow start[i] + l_i - 1$;
5 $\quad end[i] \leftarrow start[i] + |Ds_i| - 1$;
6 $\quad newSubarray[i] \leftarrow \lfloor end[i]/l_i \rfloor - \lfloor start[i]/l_i \rfloor + 1$;
7 $\quad newOverlap[i] \leftarrow \lceil (|Ds_i| - 1)/l_i \rceil$;
8 **end**
9 Transfer the cell counts of partitions among nodes to attain the *newOverlap*-sized overlaps for each chunk;
`// Now, each node independently executes the algorithm below`
10 **foreach** $C_i \in$ *chunks in the current node* **do**
11 $\quad maxCount \leftarrow$Top-1 *newSubarray*-sized subarray when the scoring function is *sum*;
12 $\quad maxDensity \leftarrow min(maxCount/(|Ds_1| \times ... \times |Ds_m|), 1)$;
13 $\quad$ Write $(i, maxDensity)$ to the current node;
14 **end**

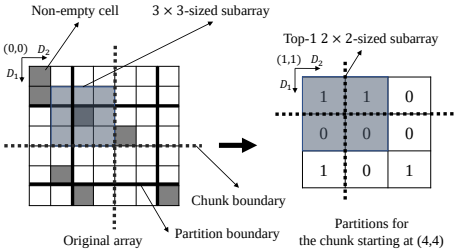


Figure 8: Maximal density estimation for the chunk starting at (4,4) in an array with four chunks.

We describe the maximal density estimation algorithm in Algorithm 4. For the sake of simplicity, we assume that the dimensions' partition size are submultiples of the corresponding dimensions' chunk sizes. In the algorithm, partitions are temporarily extended to partition boundaries, as shown in Figure 8. We introduce an example where an array is divided into four chunks in Figure 8. The given 3×3-sized subarray can overlap with maximally 2×2 partitions, as shown in the left hand side array in the figure, which means that the new subarray size becomes 2×2 (Line 6). Because the array consists of multiple chunks, the chunk overlaps must be considered. In the figure, it is shown that, to estimate the maximal density of the chunk starting at (4,4), the cell counts of partitions are transferred to obtain $\{1, 1\}$-sized chunk overlaps (Line 9). Then, the top-1 2×2-sized subarray query on the partitions with the scoring function *sum* is processed, as shown in the right hand side figure (Line 11). Because partitions of partitions are not considered, all the subarrays' scores are computed. The number of partitions is usually much smaller than that of the cells in the original array, and therefore, it is reasonable that the processing of the top-1 query is based on the naïve method. The score of the top-1 answer is 2. Then, the maximal possible density of the original subarray in the chunk of the original array is

estimated, which is 2/9 (Line 12). The UBS can be obtained from the MVS where 2/9 cells have the maximum value of a partition and other cells are empty.

7. EXPERIMENTAL EVALUATION

7.1 Experimental Setup

Settings. For the performance evaluation of the proposed methods, we used a server cluster consisting of 10 nodes with an Intel i5-2500 3.30 GHz CPU and 16 GB memory. Each node runs an instance of SciDB 18.1 on Ubuntu 14.04 LTS to play the role of a slave node in the processing of queries. The first node also performs as a master node. We configured the server cluster based on Ubuntu diskless booting. We call the time required for transferring cells to form chunk overlaps **chunk overlap transfer time** in this section. Unless otherwise noted, we used 10 as the default parameter of partition sizes for all dimensions. SciDB provides a plugin mechanism that enables users to make a user-defined operator (UDO) for their own purposes. After a UDO is loaded in SciDB, it runs as a built-in operator in SciDB. All the methods were implemented as UDOs in C++. Specifically, we used three main built-in functionalities in SciDB: (1) API to read specific chunks from disks, (2) API to transfer data (e.g., variables, chunks) among nodes, and (3) API to write specific chunks on disks. Additionally, we created a PPTS client that just returns top-k subarrays from UDOs progressively. The PPTS client was built in Python 2.7.6.

Datasets. We used three different MODIS datasets, consisting of MOD11A1 [4], MOD13Q1 [1], and MOD06L2 [3], and one synthetic dataset. MOD11A1 version 6 provides daily attributes related to land surface temperature in 1200×1200 grids [4]. Each grid represents 1 km $\times$ 1 km. We merged 12 adjacent MOD11A1 data, which resulted in 3600×4800 grids. We selected 20 days MOD11A1 data and merged them, which resulted in $20 \times 3600 \times 4800$ array having a total size of 5.9 GB. MOD13Q1 version 6 provides vegetation index values, such as the Normalized Difference Vegetation Index (NDVI) and Enhanced Vegetation Index (EVI) in 4800×4800 grids at 16-day intervals [1]. Each grid represents 0.25 km $\times$ 0.25 km. We randomly selected and merged 7 MOD13Q1 data, which resulted in a $7 \times 4800 \times 4800$ array having a total size of 12.4 GB. MOD06L2 consists of attributes related to clouds, such as cloud top pressure, cloud top height, and cloud phase [3]. We loaded MOD06L2 data for 20 days into sparse 8800×11600-sized arrays having a total size of 491 MB. MOD11A1, MOD13Q1, and MOD06L2 also include attributes related to the quality of datasets. For the experiments, we also generated synthetically a 50000×50000 two dimensional array in which an attribute $attr1$ had Gaussian distributions with different means and variances. We also added to the array an attribute $attr2$ with random values from 300 to 1000. The total size of the synthetic array data was 39.9 GB.

Query Sets. We used different scoring functions and selection conditions according to the datasets, as shown in Table 1. LST_Day_1km represents daytime land surface temperature. Clear_day_cov represents daytime clear sky coverage, and the condition avg(Clear_day_cov) $>$ 1000 means that subarrays with a very unclear sky are excluded. The NDVI measures the density of green on land using near-infrared and visible radiation. The more thickly a patch of land is wooded, the higher is the NDVI value.

Table 1: Queries for MOD11A1, MOD13Q1, MOD06L2, and the synthetic data.

	Scoring function	Selection condition
MOD11A1	sum(LST_Day_1km)	avg(Clear_day_cov) $>$ 1000
MOD13Q1	avg(NDVI)	-
MOD06L2	sum(Cloud_Top_Height)	-
Synthetic	avg(attr1)	avg(attr2) $>$ 500

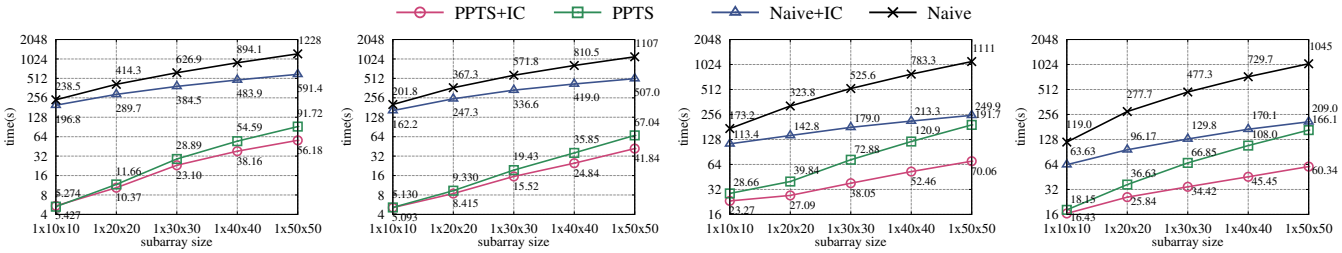
Evaluated Algorithms. In the experiments, we compared the following algorithms: (a) the naïve method (**NAÏVE**), (b) the naïve method with incremental computation (**NAÏVE + IC**), (c) progressive top-k subarray query processing without optimization (**PPTS**), (d) progressive top-k subarray query processing with incremental computation (**PPTS + IC**), and (e) progressive top-k subarray query processing with incremental computation and maximal density estimation (**PPTS + IC + ME**). We used four significant digits for the results.

7.2 Experimental Results

We first conducted experiments on real datasets, MOD11A1 and MOD13Q1 data. The chunk sizes were $1 \times 800 \times 800$ for MOD11A1 and $1 \times 500 \times 500$ for MOD13Q1, as the array data are almost equally distributed among nodes. The partitioning of the array data was performed only one time before processing queries; the times taken to partition the arrays with the partition size $1 \times 10 \times 10$ and store the results on disks for MOD11A1 and MOD13Q1 were 10.10 and 12.60 s, respectively. We performed top-k subarray queries varying the subarray size from $1 \times 10 \times 10$ to $1 \times 50 \times 50$. For the sake of efficiency of the experiments, transferring cells among nodes to form the $\{0, 49, 49\}$-sized chunk overlaps required for the queries was performed only one time before processing queries; the chunk overlap transfer times were 59.03 s and 66.73 s for MOD11A1 and MOD13Q1 data, respectively, and were excluded from query execution times.

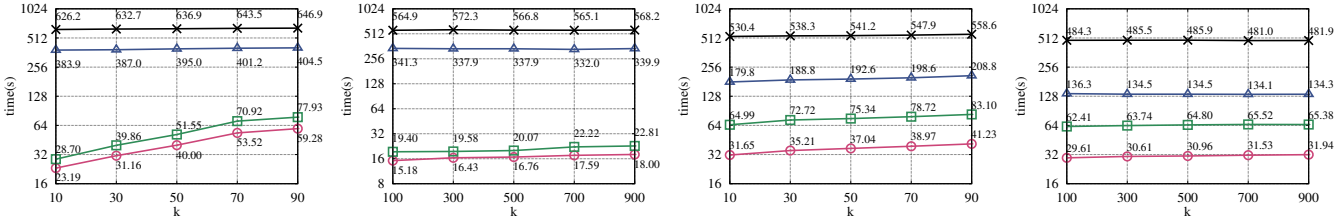
Figures 9 and 10 show that (a) PPTS outperforms NAÏVE, and PPTS+IC outperforms NAÏVE+IC in all cases, while they progressively returned answers, which cannot be achieved by the naïve methods, and (b) disjoint queries require more time to terminate than overlap-allowing queries. Figure 9 depicts the top-10 subarray query results according to subarray sizes and two subarray query types. Note that a log scale is used for the Y axis of the graphs. As the subarray size increases, all the methods require more time to complete a query, and the differences in the query execution time of PPTS and NAÏVE also increase. This indicates that the proposed methods can prune the search space efficiently, although the given subarray size increases. The naïve methods compute all the subarrays in an array in both disjoint and overlap-allowing queries; however, disjoint queries require more time because each node sends more computed subarrays with their scores to the master node. PPTS also takes more time in processing disjoint queries than overlap-allowing ones since disjoint queries usually compute more candidate subarrays to find top-k answers.

Figure 10 shows the results of top-k $1 \times 30 \times 30$-sized subarray queries according to k values and subarray query types. In all cases, the query execution times of the proposed methods are almost linear, which means that the time between the i^{th} and $(i+1)^{th} (1 \leq i < k)$ answers remains similar when the value of i is increased. In overlap-allowing



(a) MOD11A1, disjoint (b) MOD11A1,overlap-allowing (c) MOD13Q1, disjoint (d) MOD13Q1,overlap-allowing

Figure 9: Results of top-10 queries with different subarray sizes.



(a) MOD11A1, disjoint (b) MOD11A1,overlap-allowing (c) MOD13Q1, disjoint (d) MOD13Q1,overlap-allowing

Figure 10: Results of top-k $1 \times 30 \times 30$-sized subarray queries with different k.

Table 2: Average/standard deviation of delays between the i^{th} and $(i+1)^{th}(1 \leq i < k)$ answers in Figure 10. (seconds)

Methods	MOD11A1, disjoint	MOD13Q1, disjoint
PPTS + IC	0.4844/0.8054	0.1317/0.1489
PPTS	0.6287/1.024	0.2383/0.2656

queries, if the method selects a subarray as one of the top-k subarrays, it tends also to select other subarrays near that selected as one of the top-k subarrays, which results in a similar query execution time, although k increases, and less meaningful top-k subarrays. When k is varied, the naïve method takes almost the same time to complete queries in overlap-allowing queries. The time complexities of the naïve methods in overlap-allowing and disjoint queries are $O(nlogk)$ and $O(nlogn + nk)$, respectively, where n is the number of cells in the array. Therefore, if the value of n is large and k value is relatively small, the time taken to complete queries does not change drastically according to k in the naïve methods. However, in disjoint queries, the time increases more faster than in overlap-allowing queries when k value grows because the slave nodes send more computed subarrays to the master node. The top-90 disjoint queries in Figure 10(a) and Figure 10(c) actually mean the top-49146 and 54531 overlap-allowing query, respectively.

In Figure 9 and 10, it can be seen that the incremental computation is efficient, especially when the subarray size and the k value are large. However, the optimization technique cannot be applied to queries with holistic scoring functions. Although PPTS outperformed NAÏVE in the experiments, if attribute values of an array get more and more similar, PPTS has difficulty pruning the search space. It is a common limitation in threshold-based algorithms.

Figure 11 shows communication costs during query processing including the chunk overlap transfer time. In PPTS, Node 1, the master node, utilizes the network when deciding the $i^{th}(1 \leq i \leq k)$ answer. Specifically, Node 1 requests the next local answers to maximally $N-1$ slave nodes, receives the local answers from the nodes, and sends the i^{th} answer to the nodes, which means that the total number of communications is maximally $(N-1) \times 3 \times k$ during query processing. Because we configured our server cluster based

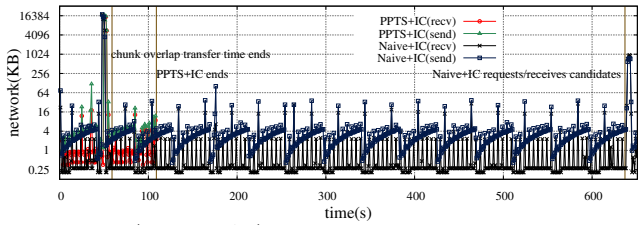
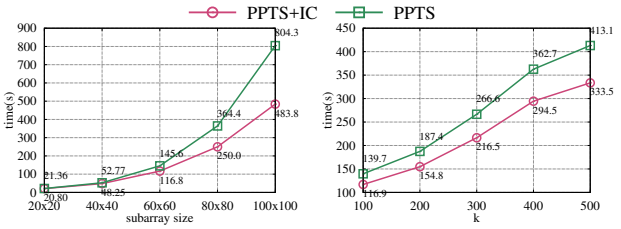


Figure 11: (MOD11A1) Communication costs in Node 1 with the disjoint top-10 $1 \times 50 \times 50$-sized subarray query.

on diskless booting, small communication costs were consistently required to maintain the system. The same patterns were observed in the MOD13Q1 dataset.

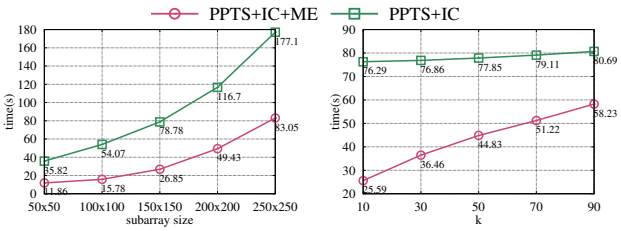
Figure 12 shows the disjoint top-k subarray query results on the synthetic data with various subarray sizes and k values. The naïve method always failed, because of the "out of memory" error, and therefore, we excluded it from the figure. The chunk size was set as 1500×1500, and one-time partitioning took 179.0 s. The $\{99, 99\}$-sized chunk overlaps that are required for the subarray query with the largest size (100×100) in the experiments were pre-computed and stored before processing queries. In Figure 12(a), it can be seen that the query execution times rapidly increase as the subarray sizes increase. This is because (1) the cost of computing a subarray increases and (2) outstanding regions with high attribute values are covered by a smaller number of subarrays; thus, for queries with a larger subarray size, it is necessary to find more subarrays in the data than for those with a smaller subarray size. Figure 12(b) shows that the delays between consecutive answers remain almost constant, irrespective of k. The average values of delays for PPTS + IC and PPTS were 0.6479 s and 0.7432 s, respectively.

We conducted experiments on the UBS optimization based on maximal density estimation proposed in Section 6.2 in the case of MOD06L2 data (Figure 13). We ran queries on 20 independent array datasets, and averaged the results. The chunk size was set as 1500×1500. One-time partitioning took 0.6794 s on average. When the subarray size was 250×250, the average chunk overlap transfer time was 2.235 s and the average time to estimate maximal densities was 2.034 s. The results in the graphs include



(a) $k = 50$ (b) 50×50

Figure 12: Disjoint top-k queries on synthetic data.



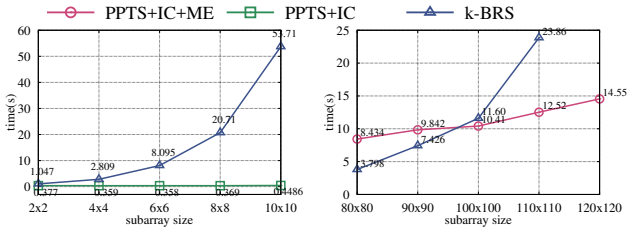
(a) $k = 10$ (b) 150×150

Figure 13: Effects of maximal density estimation on MOD06L2 dataset and disjoint top-k queries.

the times taken to perform maximal density estimation and exclude the chunk overlap transfer times. Without maximal density estimation, partitions' UBSs are significantly higher than the scores of actual subarrays overlapping with partitions, because the maximal virtual subarrays assume that they have the maximum values in all cells without empty cells, although MOD06L2 data constitute sparse arrays. This means that, even when one of the top-k subarrays is found in a partition, the query cannot satisfy the answer-returning condition and has to continue query processing to the subsequent partitions. On the basis of maximal density estimation, the query terminates much earlier, as shown in the figures.

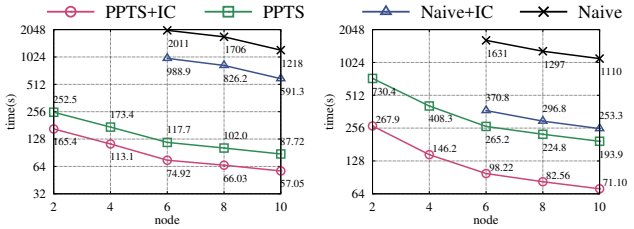
Figure 14 shows comparisons between PPTS and k-BRS [21] on disjoint top-10 subarray queries with various subarray sizes in a single node. We randomly selected 5 arrays from MOD11A1 and MOD06L2 datasets, respectively. We ran the queries in each array and averaged the results. For MOD11A1 datasets, we extracted the 100×100 subarray starting at $(0, 0)$ from each array (3600×4800) as k-BRS always failed on the original MOD11A1 datasets because of memory overflow. The average percentage of non-empty cells were 83.04% (MOD11A1) and 0.08478% (MOD06L2). k-BRS can be executed only when a whole array forms a single chunk. For PPTS, we used one chunk in MOD11A1 and 1000×1000 chunking in MOD06L2. In Figure 14(b), the results of PPTS+IC+ME included the chunk overlap transfer times and maximal density estimation times. For PPTS, partition sizes were 2×2 and 10×10, and average partitioning times were 0.4978 s and 1.655 s in MOD11A1 and MOD06L2, respectively.

As shown in Figure 14, k-BRS was highly inefficient in MOD11A1 datasets while it performed better than PPTS in MOD06L2 datasets only on relatively small subarray sizes. This is because the sweep algorithm of k-BRS quickly decides the region that includes some objects. However, as the subarray size increased, the query execution times of k-BRS increased rapidly, and it eventually failed on the subarray size of 120×120 because of memory overflow caused by investigating a huge number of candidates in MOD06L2. In addition to better performance, PPTS also supports the



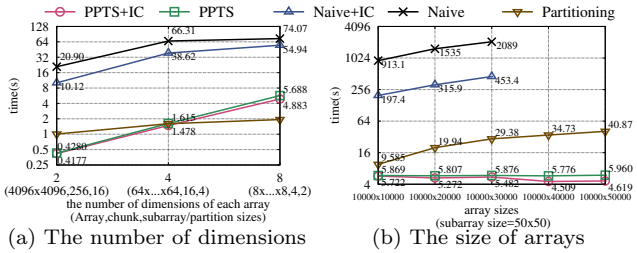
(a) MOD11A1 (dense) (b) MOD06L2 (sparse)

Figure 14: Comparisons between PPTS and k-BRS (disjoint top-10 queries without selection conditions, single node).



(a) MOD11A1 (b) MOD13Q1

Figure 15: Disjoint top-10 $1 \times 50 \times 50$-sized subarray queries varying the number of nodes.



(a) The number of dimensions (b) The size of arrays

Figure 16: Disjoint top-10 subarray queries varying the number of dimensions and the size of arrays.

wider range of scoring functions and distributed processing based on the array data model.

Figure 15 shows disjoint top-10 subarray query results on the real datasets varying the number of nodes. When the number of nodes was 2 or 4, the naïve methods failed because of the "out of memory" error. PPTS benefited from the increasing number of nodes in both datasets; however, the straggler problem cannot be avoided. This is because (a) how the array data are distributed among nodes affects query processing and (b) PPTS is a synchronous algorithm.

Figure 16 shows the disjoint top-10 subarray query results varying the number of dimensions and the size of arrays. Each array contained 30 subarrays with random attribute values in the range of $[600, 1000]$, and the other cells had random values in $[0, 500]$. As the number of dimensions increases (Figure 16(a)), overhead of computing subarrays also increases because the number of consecutively accessible cells, the most minor dimension size, decreases. The reason also explains the decreasing impact of IC in the naïve methods. Because the $4 - d$ array was distributed somewhat unequally with the chunk size, the naïve methods operated inefficiently. Figure 16(b) shows that PPTS efficiently prunes the search space regardless of the size of original arrays, which means that PPTS is scalable.

7.3 Effects of Parameters

To evaluate the effect of a partition size on the performance of the proposed method, we conducted experi-

Table 3: Effects of partition sizes on MOD11A1 and MOD13Q1 with the chunk size of 800 × 800. (seconds,KB)

Partition size	10	50	100	150	200	250	300	350
MOD11A1								
PPTS + IC	21.31	24.55	27.57	28.30	30.98	43.52	40.13	42.50
Partitioning time	1.675	1.038	1.010	1.010	1.001	1.033	1.006	1.039
Required storage	2811	162.2	49.30	33.54	18.87	19.92	14.93	14.95
MOD13Q1								
PPTS + IC	36.9	42.0	47.0	60.9	53.8	59.3	65.1	65.4
Partitioning time	4.022	2.578	2.490	2.581	2.488	2.661	2.568	2.705
Required storage	8435	356.8	95.58	59.72	30.97	32.31	22.02	22.24

ments using 20 independent MOD11A1 and 15 independent MOD13Q1 array datasets with various partition sizes. We ran disjoint top-10 subarray queries with a subarray size of 50×50 on each array using PPTS + IC and averaged the results. The effects of the partition sizes are shown in Table 3. The partition size given in the table was applied to all dimensions. The partitioning time includes the time required to read arrays from disks, create partitions, and write partitions on disks. The required amount of storage for partitions is also shown in the tables. The results indicate that, as the partition size increases, the proposed method takes a longer time to complete the queries. This is because, although the queries already find one of the top-k subarrays, they cannot terminate, as there remain subarrays to be computed that overlap with the current partition. In contrast, the amount of storage for partitions decreases as the partition size increases because of the smaller number of partitions. The time complexity of partitioning is $O(n + NlogN)$, where n is the number of cells in an array and N is the number of partitions. The value of N was considerably smaller than that of n in these experiments. Therefore, it was observed that the partitioning time was independent of the size of the partitions except for when the partition size was 10. If the partition size is extremely small, the time complexity approaches more closely to $O(nlogn)$, and the required storage increases rapidly. Sometimes there existed cases that it took longer time to terminate queries even with the smaller size of partitions. This was because, for example, if a region with high attribute values was covered by not one partition A but several partitions B with the smaller partition size, the coverage of B could be wider than the region and A, and more partitions had to be checked; more subarrays also had to be computed. In summary, if an array is partitioned using small partition sizes, the top-k subarrays can be obtained earlier; however, the partitions require a larger amount of storage space.

8. RELATED WORK

Array Databases. Array database management systems have been studied under the necessity of managing multi-dimensional data. RasDaMan [6] was the first fully operational array database. ArrayStore [22] supports parallel processing and various workloads for arrays. SciDB [7, 23] constitutes the de facto standard open source array database with steady version updates. TileDB [20] is an array storage manager and provides embeddable libraries to efficiently manage both dense and sparse arrays, providing fast updates. ChronosDB [28] is a distributed and geospatial array database with command line tools; however, it is still under development.

Data Exploration in Array Databases. Searchlight [16, 17] provides efficient subarray exploration by integrating constraint programming solvers and main-memory synopses. It can be considered an efficient filter for subarrays that satisfy the selection conditions given in our study. Our approach focuses mainly on finding the top-k subarrays with regard to a scoring function, and hence allows a different type of data exploration in array databases.

Top-k Query Processing. Top-k queries have been extensively studied in various fields. In [14], the top-k query types and processing using various aspects in relational databases were summarized. Fagin [11] proposed the Threshold Algorithm (TA) that retrieves top-k answers by setting threshold values. To apply TA to a top-k subarray query, we have to compute and sort all subarray's scores, which means the naïve method. TA mainly targets scoring functions that consider several attributes together while PPTS considers the measure attribute. Top-k region queries have been studied in spatial databases. In [18], the top-k hot region queries that find k non-overlapping regions with regard to a subject were defined. A region is scored by the weighted sum of the number of objects. The MaxRS problem [9] was proposed for finding the top-1 rectangle of a given size where spatial objects' weighted sum is maximized. Best Region Search [12] finds the top-1 region given a set of spatial objects and the size of a two dimensional rectangle based on a submodular monotone scoring function. The k-BRS problem [21] finds the top-k best rectangles according to a monotone scoring function. Based on the vertical/horizontal sweep algorithm, k-BRS finds slabs and regions. These approaches [9, 12, 18, 21] do not consider the array data model, and they support limited scoring functions. In particular, the approaches presented in [9, 12] target only the top-1 region. Furthermore, all of them do not provide distributed processing, which is critical for large scale data analysis.

In addition, there exist studies on distributed threshold-based top-k query processing [5, 8, 19, 25, 26, 27]. In DiTo [26, 27], each node stores the skyline objects for its own data as a summarization, and a coordinator node processes top-k queries progressively based on the summarizations. TPUT [8] and KLEE [19] prune ineligible objects based on threshold values while minimizing network traffic. In [5], a study on distributed top-k query processing considering keywords was reported. RIPPLE [25] is a generic framework for processing rank queries such as top-k and skyline queries using distributed hash tables. The results of these studies cannot be directly applied to top-k subarray query processing, because they did not consider the concept of subarrays.

9. CONCLUSION

In this paper, we introduced progressive top-k subarray query processing. In our method, first an array is partitioned and important information is extracted from partitions. Second, the proposed PPTS method based on partitions, which is further extended to distributed processing, is applied. Then, we devised optimization techniques to improve PPTS. Finally, we showed by extensive experiments that PPTS outperforms the existing method to a noteworthy extent and is effective over large real and synthetic array data.

Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2017R1A2A2A05069318).

10. REFERENCES

[1] Didan, K.. MOD13Q1 MODIS/Terra Vegetation Indices 16-Day L3 Global 250m SIN Grid V006. 2015, distributed by NASA EOSDIS LP DAAC, https://doi.org/10.5067/MODIS/MOD13Q1.006.

[2] NASA VISIBLE EARTH. https://visibleearth.nasa.gov/view.php?id=92137.

[3] Platnick, S., Ackerman, S., King, M., et al., 2015. MODIS Atmosphere L2 Cloud Product (06_L2). NASA MODIS Adaptive Processing System, Goddard Space Flight Center, USA: http://dx.doi.org/10.5067/MODIS/MOD06_L2.006.

[4] Wan, Z., S. Hook, G. Hulley. MOD11A1 MODIS/Terra Land Surface Temperature/Emissivity Daily L3 Global 1km SIN Grid V006. 2015, distributed by NASA EOSDIS LP DAAC, https://doi.org/10.5067/MODIS/MOD11A1.006.

[5] D. Amagata, T. Hara, and S. Nishio. Distributed top-k query processing on multi-dimensional data with keywords. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, page 10. ACM, 2015.

[6] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *Acm Sigmod Record*, volume 27, pages 575–577. ACM, 1998.

[7] P. G. Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 963–968. ACM, 2010.

[8] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 206–215. ACM, 2004.

[9] D.-W. Choi, C.-W. Chung, and Y. Tao. A scalable algorithm for maximizing range sum in spatial databases. *PVLDB*, 5(11):1088–1099, 2012.

[10] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, S. Madden, M. Stonebraker, S. B. Zdonik, and P. G. Brown. Ss-db: A standard science dbms benchmark. *Under submission*, 2010.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.

[12] K. Feng, G. Cong, S. S. Bhowmick, W.-C. Peng, and C. Miao. Towards best region search for data exploration. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1055–1070. ACM, 2016.

[13] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, 1(1):29–53, 1997.

[14] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.

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

[16] A. Kalinin, U. Cetintemel, and S. Zdonik. Searchlight: Enabling integrated search and exploration over large multidimensional data. *PVLDB*, 8(10):1094–1105, 2015.

[17] A. Kalinin, U. Cetintemel, and S. Zdonik. Interactive search and exploration of waveform data with searchlight. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2105–2108. ACM, 2016.

[18] J. Liu, G. Yu, and H. Sun. Subject-oriented top-k hot region queries in spatial dataset. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 2409–2412. ACM, 2011.

[19] S. Michel, P. Triantafillou, and G. Weikum. Klee: A framework for distributed top-k query algorithms. In *Proceedings of the 31st international conference on Very Large Data Bases*, pages 637–648. VLDB Endowment, 2005.

[20] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson. The tiledb array data storage manager. *PVLDB*, 10(4):349–360, 2016.

[21] D. Skoutas, D. Sacharidis, and K. Patroumpas. Efficient progressive and diversified top-k best region search. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 299–308. ACM, 2018.

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

[23] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The architecture of scidb. In *International Conference on Scientific and Statistical Database Management*, pages 1–16. Springer, 2011.

[24] R. Taft, M. Vartak, N. R. Satish, N. Sundaram, S. Madden, and M. Stonebraker. Genbase: A complex analytics genomics benchmark. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 177–188. ACM, 2014.

[25] G. Tsatsanifos, D. Sacharidis, and T. K. Sellis. Ripple: A scalable framework for distributed processing of rank queries. In *EDBT*, pages 259–270, 2014.

[26] A. Vlachou, C. Doulkeridis, and K. Nørvåg. Distributed top-k query processing by exploiting skyline summaries. *Distributed and Parallel Databases*, 30(3-4):239–271, 2012.

[27] A. Vlachou, C. Doulkeridis, K. Nørvåg, and M. Vazirgiannis. On efficient top-k query processing in highly distributed environments. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 753–764. ACM, 2008.

[28] R. A. R. Zalipynis. Chronosdb: distributed, file based, geospatial array dbms. *PVLDB*, 11(10):1247–1261, 2018.