International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 3, Number 2, pages 244–263, July 2013

Distributed Spatial k-dominant Skyline Maintenance Using Computational Object Preservation

Md. Anisuzzaman Siddique, Asif Zaman, Md. Mahbubul Islam
Computer Science and Engineering Department
University of Rajshahi
Rajshahi-6205, Bangladesh
Email: anis_cst@yahoo.com, asifzaman3180@yahoo.com, mmi.cse@gmail.com

and

Yasuhiko Morimoto
Graduate School of Engineering
Hiroshima University
Kagamiyama 1-7-1, Higashi-Hiroshima 739-8521, Japan
Email: morimoto@mis.hiroshima-u.ac.jp

Received: February 20, 2013
Revised: May 14, 2013
Accepted: June 18, 2013
Communicated by Sayaka Kamei

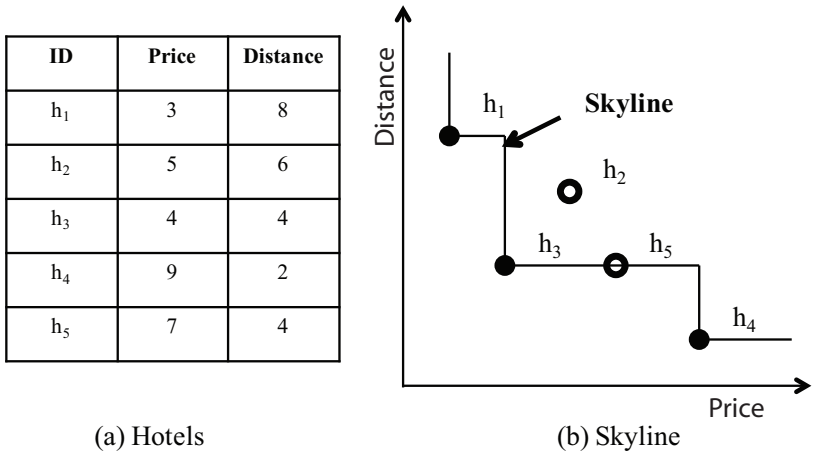
Abstract

The existing k-dominant skyline solutions are restricted to centralized query processors, limiting scalability, and imposing a single point of failure. To overcome those problems in this paper, we propose the computation and maintenance algorithms for spatial k-dominant skyline query processing in large-scale distributed environment. Where the underlying dataset is partitioned into geographically distant computing core (personal computer) that are connected to the coordinator (server). Our proposed techniques preserve the spatial k-dominant computation object itself into a serialized form. This preservation is done in client's core after completing a computational job successfully. When the issue of maintenance comes in action, preserve data object retrieves and use for computation. This procedure eliminates the necessity of intermediate re-send and re-computation of k-dominant skyline for the maintenance issue. Thus, we quantify the gain of data transferring consecutively into different cores to maximize the overall gain as well as the query or balancing the load on different cores fairly. Extensive performance study shows that proposed algorithms are efficient and robust to different data distributions.

Keywords: Skyline, k-dominant Skyline, Load table, Maintenance

1 Introduction

An user often wants to optimize their decision making and selection criteria across multiple attributes. In many cases, especially in multi-criteria decision making, there is no single best answer to a query. Skyline queries do not require an explicit preference function, which may be difficult for the user to define when the relative importance of the different criteria is vague.



ID	Price	Distance
h_1	3	8
h_2	5	6
h_3	4	4
h_4	9	2
h_5	7	4

(a) Hotels (b) Skyline

Figure 1: Skyline Example

A skyline query retrieves a set of skyline objects so that the user can choose promising objects from them and make further inquiries. Skyline objects in a database are objects that are not dominated by any other objects in the database. Given an n-dimensional database DB, an object O_i is said to be in skyline of DB if there is no other object O_j $(i \neq j)$ in DB such that O_j is better than O_i in all dimensions. If there exists such O_j, then we say that O_i is dominated by O_j or O_j dominates O_i. Figure 1 shows a typical example of skyline. The table in the figure is a list of hotels, each of which contains two numerical attributes: *distance* and *price*, for online booking. A user chooses a hotel from the list according to her/his preference. In this situation, her/his choice usually comes from the hotels in skyline, i.e., one of h_1, h_3, h_4 (see Figure 1 (b)). Therefore, such skyline query functions are important for several database applications, including customer information systems, decision support, data visualization, and so forth. A number of efficient algorithms for computing skyline objects have been reported in the literature [1, 2, 3, 4, 5].

In a high-dimensional dataset a skyline query often retrieves too many objects to investigate. To minimize the result and to find more important and meaningful objects, Chan et al. considered k-dominant skyline query [6]. They relaxed the definition of "dominated" so that an object is likely to be dominated by another. Given an n-dimensional database, an object O_i is said to k-dominates another object O_j $(i \neq j)$ if there are k $(k \leq n)$ dimensions in which O_i is better than or equal to O_j. A k-dominant skyline object is an object that is not k-dominated by any other objects.

Until now the k-dominant skyline literature has focused on centralized databases. In practice, data are often collected from multiple sources that are geographically distant from each other. For example, assume a travel agency has numerous computing cores (data sources), each of which is located in a different city (New York, Los Angeles, Chicago, etc.), and manages only local properties. A query may demand the k-dominant skyline of the properties in specific source via coordinator (server). For this or similar type application, a distributed architecture is more suitable. This is because, compared to the centralized counterpart, it has smaller computation cost, it allows better local data management, smaller update cost, and higher tolerance to machine failures.

In this paper, we study the efficient maintenance of spatial k-dominant skyline queries, where the underlying dataset is partitioned into geographically distant computing cores that are connected to the coordinator. Assume a system architecture, that consists of a server with a network access point, and several core connect to the server (Figure 2). The server can directly communicate with any core and request for k-dominant skyline computation. In such a setting, each time the server sends a k-dominant skyline request with new dataset or maintenance request to a core, all objects information that are not k-dominated have to be transferred to the server. Our previous work [16] suffers from data transmission point of view. The server needs to resend the dataset again and again for each update. For example, assume initially cores C_1, C_2, and C_3 are assigned to process three regional datasets R_1, R_2, and R_3 respectively. After completing the computational work when a

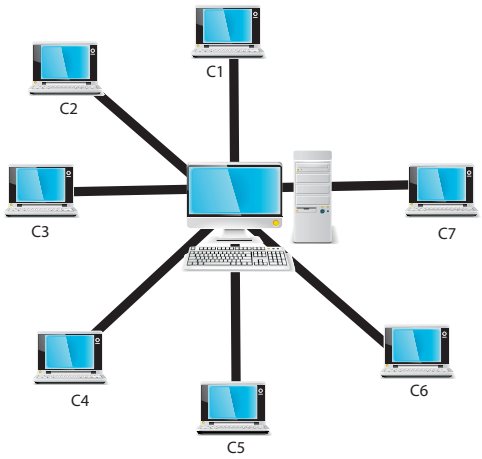


Figure 2: System architecture

core C_1, becomes free then the new dataset D_4 will be assigned to it. In our previous work [16], for computational complexity, C_1 did not store internal data structure related to D_1. Next, when dataset D_1 requires some maintenance work then the server needs to assign a new core to perform the same computation and maintenance operation respectively. This procedure consumes a lot of computing power. The proposed work overcomes that problem and the controlling server need not to resend the dataset again and again for each maintenance issue. In this approach data computation object itself is stored as *binary logic object data* in the corresponding computing core. This technique usually used to store image objects. As computation object itself is stored, we don't have to worry about the complexity of storing internal data structure related to a dataset. Computational objects are stored and can be restored again if necessary.

Wu et al. [29] proposed distributed skyline query processing algorithm (DSL), to support skyline search on a horizontally partitioned dataset. DSL is not applicable to our problem, where the underlying database DB is horizontally partitioned onto the participating cores in an arbitrary manner. Specifically, we allow each core to contain any subset of DB, rather than only the data falling in a particular region. It is worth mentioning that the partitioning scheme adopted by DSL incurs expensive network overhead for maintenance DB. For example, assume that core C_1 receives an insertion to DB that, however, falls in the region assigned to core C_2. Since the object must be retained at core C_2, an object transfer (from core C_1 to core C_2) is required. In general, every insertion may be accomplished by an object transfer, which is prohibitively costly in practice. Similar thing will happen for delete as well as update operations. The proposed work does not suffer from these drawbacks, because each core can manage its local dataset without any network transmission.

Here are the contribution of this paper:

(1) We extend k-dominant computation from centralized to distributed datasets. We propose a framework for the maintenance of distributed spatial k-dominant skyline queries over multiple cores, aiming to reduce data transmission overhead.

(2) We introduce three novel algorithms for distributed k-dominant skyline queries. The registration and ranking power calculation, and the job distributed algorithms work in server site. For client site we develop computation core selection algorithm.

(4) Moreover, all computing cores are reliable, failure of one or more computing core does not create any problem, the rest of the computing cores are responsible to conduct the computational work.

(5) We conduct extensive experiments to perform the evaluation. The evaluation results show that the proposed techniques are efficient and scalable. Our algorithms consistently outperform against its competitor *multicore based spatial k-dominant skyline* (MSKS) in all examined setups.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the notions and properties of regional k-dominant skyline computation. We provide detailed

examples and analysis of our algorithm in Section 4. We experimentally evaluate our algorithm in Section 5 under a variety of settings. Finally, Section 6 concludes the paper.

2 Related Work

Our work is motivated by previous studies of skyline query processing, k-dominant skyline query processing as well as distributed k-dominant skyline query processing, which are reviewed in this section.

2.1 Skyline Query Processing

Borzsonyi et al. first introduced the skyline operator over large databases and proposed three algorithms: *Block-Nested-Loops*(BNL), *Divide-and-Conquer*($D\&C$), and B-tree-based schemes [2]. BNL compares each object of the database with every other object, and reports it as a result only if any other object does not dominate it. A window W is allocated in main memory, and the input relation is sequentially scanned. In this way, a block of skyline objects is produced in every iteration. In case the window saturates, a temporary file is used to store objects that cannot be placed in W. This file is used as the input to the next pass. $D\&C$ divides the dataset into several partitions such that each partition can fit into memory. Skyline objects for each individual partition are then computed by a main-memory skyline algorithm. The final skyline is obtained by merging the skyline objects for each partition. Chomicki et al. improved BNL by presorting, they proposed *Sort-Filter-Skyline*(SFS) as a variant of BNL [4]. Among index-based methods, Tan et al. proposed two progressive skyline computing methods Bitmap and Index [7]. In the Bitmap approach, every dimension value of a point is represented by a few bits. By applying bit-wise AND operation on these vectors, a given point can be checked if it is in the skyline without referring to other points. The index method organizes a set of d-dimensional objects into d lists such that an object O is assigned to list i if and only if its value at attribute i is the best among all attributes of O. Each list is indexed by a B-tree, and the skyline is computed by scanning the B-tree until an object that dominates the remaining entries in the B-trees is found. The current most efficient method is *Branch-and-Bound Skyline*(BBS), proposed by Papadias et al., which is a progressive algorithm based on the *best-first nearest neighbor (BF-NN)* algorithm [5]. Instead of searching for nearest neighbor repeatedly, it directly prunes using the R*-tree structure.

Recently, more aspects of skyline computation have been explored. Vlachou et al. introduce the concept of extended skyline set, which contains all data elements that are necessary to answer a skyline query in any arbitrary subspace [9]. Fotiadou et al. mention about the efficient computation of extended skylines using bitmaps in [20]. Chan et al. introduce the concept of *skyline frequency* to facilitate skyline retrieval in high-dimensional spaces [11]. Tao et al. discuss skyline queries in arbitrary subspaces [12].

2.2 k-Dominant Skyline Query Processing

Chan et al. introduced k-dominant skyline query [6]. They proposed three algorithms, namely, *One-Scan Algorithm (OSA)*, *Two-Scan Algorithm (TSA)*, and *Sorted Retrieval Algorithm (SRA)*. OSA uses the property that a k-dominant skyline objects cannot be worse than any skyline object on more than k dimensions. This algorithm maintains the skyline objects in a buffer during the scan of the dataset and uses them to prune away objects that are k-dominated. TSA retrieves a candidate set of dominant skyline objects in the first scan by comparing every object with a set of candidates. The second scan verifies whether these objects are truly dominant skyline objects or not. This method turns out to be much more efficient than the one-scan method. A theoretical analysis is provided to show the reason for its superiority. The third algorithm, SRA is motivated by the rank aggregation algorithm proposed by Fagin et al., which pre-sorts data objects separately according to each dimension and then merges these ranked lists [8].

The above skyline as well as k-dominant methods are designed for centralized database and do not work in the distributed environment. Several attempts have been made to address distributed

skyline retrieval, leading to a number of interesting methods [17, 14]. Balke et al. extended the skyline problem for the web in which the attributes of an object are distributed in different web-accessible servers. They assumes that a d-dimensional database is vertically partitioned into d lists, where each list contains the values of a dimension in ascending order. They also develop an enhanced solution that, instead of visiting the lists in a round-robin fashion, always accesses the most promising list, so that the least expansion is performed on each list to discover an anchor point p. Huang et al. [14] study skyline search on hand-held devices in a mobile ad-hoc network (MANET). Each device store a tiny horizontal fraction of the underlying relation, and is able to communicate only with its *neighbors*, i.e., devices within its contact range. The connection is ad-hoc because (i) two devices cease to be neighbors when they move out of each other's contact range, and (ii) new neighbors are formed when two devices come close to each other. The objective is to compute the skyline over the data of all the devices and report it at the querying device Q, by consuming as little bandwidth as possible.

2.3 Distributed k-Dominant Skyline

Distributed skyline computation has recently attracted considerable attention and has been studied in a variety of distributed systems, including web information systems [17], parallel systems [23], peer-to-peer systems [18, 20, 21, 24, 25, 27, 28, 29], mobile ad-hoc networks [26], as well as more generic distributed systems [19, 22]. Most of the existing approaches focus on highly distributed environments, such as peer-to-peer networks, assuming that all data sources store common attributes. Several approaches belong to this category, namely DSL [29], SSP [27], SKYPEER [24], and BITPEER [20]. DSL was proposed by Wu et al. [29] and it is the first paper that addresses constrained skyline query processing over disjoint data partitions by using a structured peer-to-peer overlay, namely CAN. Wang et al. [27] propose the SSP algorithm based on the use of a tree based peer-to-peer overlay (BATON) for assigning data to servers. SKYPEER [24] transforms the multi-dimensional data into one-dimensional values and utilizes a thresholding scheme in order to reduce the transferred data. Fotiadou et al. [20] propose BITPEER for supporting efficiently continuous subspace skylines in a distributed setting by using distributed bitmap indexes. However, unfortunately none of the above methods are designed for distributed k-dominant skyline computation. This is because the maintenance of k-dominant skyline for an update is much more difficult due to the intransitivity of the k-dominance relation. Assume that "A" k-dominates "B" and "B" k-dominates "C". However, "A" does not always k-dominate "C". Moreover, "C" may k-dominate "A". Because of the intransitivity property, we have to compare each object against every other object to check the k-dominance. Therefore in [15], Siddique and Morimoto, resolved k-dominant skyline maintenance problem in a centralized dataset. We proposed $MSKS$ [16] algorithm for spatial k-dominant skyline computation and maintenance. The server needs to resend the dataset again and again for each update. In this paper, we expand the k-dominant skyline queries to spatially distributed datasets and resolve the dataset resending problem.

3 Preliminaries

This section discusses the distributed k-dominant skyline computation, maintenance problems, and associated properties. Our objective is to extract the local k-dominant skyline and deliver it to a server computer CR. We refer CR as the *coordinator* of the k-dominant skyline retrieval. Assume there are x regional datasets named $R_1, R_2, \cdots, R_x$ and y computing cores called $C_1, C_2, \cdots, C_y$. CR can be any computer other than $C_1, C_2, \cdots, C_y$. CR is able to initiate communication with each core directly. Assume each dataset is n-dimensional and $D_1, D_2, \cdots, D_n$ are the n attributes of each regional dataset. Let O_1, O_2, $\cdots$, O_r be r objects (tuples) of a regional dataset, say R_p $(1 \leq p \leq x)$. We use $O_i.D_j$ to denote the j-th dimension value of O_i.

3.1 k-Dominance

An object O_i is said to *dominate* another object O_j, which we denote as $O_i \leq O_j$, if $O_i.D_s \leq O_j.D_s$ for all dimensions D_s $(s = 1, \cdots, n)$ and $O_i.D_t < O_j.D_t$ for at least one dimension D_t $(1 \leq t \leq n)$. We call such O_i as *dominant object* and such O_j as *dominated object* between O_i and O_j.

By contrast, an object O_i is said to *k-dominate* another object O_j, denoted as $O_i \leq_k O_j$, if $O_i.D_s \leq O_j.D_s$ in k dimensions among n dimensions and $O_i.D_t < O_j.D_t$ in one dimension among the k dimensions. We call such O_i as *k-dominant object* and such O_j as *k-dominated object* between O_i and O_j.

An object O_i is said to have *δ-domination power* if there are δ dimensions in which O_i is better than or equal to all other objects of R_p.

3.2 Spatial k-Dominant Skyline

An object $O_i \in R_p$ $(1 \leq p \leq x)$ is said to be a *spatial skyline object* of R_p if O_i is not dominated by any other object in R_p. Similarly, an object $O_i \in R_p$ is said to be a *spatial k-dominant skyline object* of R_p if O_i is not k-dominated by any other object in R_p. We denote a set of all spatial k-dominant skyline objects in R_p as $Sky_k(R_p)$. Note that objects that have k-domination power must be k-dominant skyline objects but not vice versa.

4 Distributed Spatial k-dominant Skyline

In this section, we present our algorithms for job distributing and computing spatial k-dominant skyline objects and maintaining the result when update occurs. There is a crucial difference between our network architecture and those in [14, 9, 29]. The architecture in [14, 9, 29] aim at supporting massive distributed environments with a huge number of data machines. In our scenario the cores number is moderately small.

Our job distribution algorithms are useful for efficient k-dominant skyline retrieval in distributed environment. Job distribution is important for three reasons. (i) Efficient distributing method enhances user experience by preventing a long idle period. (ii) It allocates the best computing core to the largest regional dataset. (iii) It provides core allocation guarantee for every regional dataset. We illustrate job distributing technique among multiple cores in Section 4.1. Then show how to compute spatial k-dominant skyline for all k at a time in Section 4.2. Section 4.3 presents three types of maintenance solution. They are insertion, deletion, and update operation.

The distributed system is designed in such a manner for enhanced computation speed. If a user tries to compute in single core, it will consume a lot of computing time as well as computing power. The user may complain about the overhead of transmitting full dataset each time a new computation is initiated. The proposed techniques do not suffer from transmission issue.

4.1 Job Distribution

If the number of regional dataset is equal to or smaller than the available computing core (i.e., $x \leq y$) then there is no issue to consider for job distribution. However, when $x > y$, (i.e., the number of regional dataset is larger than the available computing core) job distribution issues occur. We propose three algorithms for job distribution. They are: registration and ranking power calculation, job allocation, and computation core selection. Among the algorithms registration and ranking power calculation and job allocation algorithms work in server site and the remaining computation core selection algorithm works in client site.

Registration and Ranking Power Calculation

Suppose there are y computing cores called $C_1, C_2, \cdots, C_y$ for distributed spatial k-dominant skyline computation. Every node needs to register itself to the coordinator CR. The CR has an independent thread for registering and ranking computing cores. A core (say C_m) sends a registration request

Algorithm 1: Registration and Ranking Power Calculation

Create LoadTable(ID, IP Address, Port no., Computing power, Memory size, Network traffic, Rank power, Status)
1. Start and wait 100ms for a registration request.
2. If *new request* arrives then
   i. Assign a new ID I_m for requesting core C_m.
   ii. Store C_m's $C_m(IP)$, $C_m(CP)$, $C_m(M)$ in standard database
   iii. Set I_m status as "FREE".
   Else
   i. Select all cores whose status are not BUSY (i.e., either "FREE" or "DEAD").
   ii. For each core C_m do the following steps:
   a) Ping each $C_m(IP)$ 4-6 times and keep average response time $C_m(RT)$.
   b) If $C_m(RT)$ = infinity, i.e., destination host is unreachable, then
   Set C_m status as "DEAD" and continue.
   c) Calculate C_m ranking power, $RP_m = 6 * C_m(CP) + C_m(M) + (C_m(RT)^{-2})$.
   d) Store RP_m in LoadTable.
3. Go to step 1.

to CR. Then the independent thread of CR creates a load table named LoadTable(ID, IP Address, Port no., Computing power, Memory size, Network traffic, Rank power, Status). ID can be any value between 1 to y. Each core status can be either "FREE", "BUSY" or "DEAD". Computing power $C_m(CP)$, memory size $C_m(M)$ are supplied by corresponding core during the registration procedure. Proposed method periodically monitors the traffic status during RP_m calculation. The average ping time or response time $C_m(RT)$, from controlling core CR to C_m uses as an indication of traffic status. If the network is busy, the average ping time will be high. Again if C_m is disconnected then the average ping time becomes infinity. In such situation C_m is marked as a dead core, i.e., core status field of LoadTable for C_m is set as "DEAD". We consider three parameters such as computing power, memory size, and network traffic to calculate RP_m of C_m. $C_m(CP)$ and $C_m(M)$ are provided by C_m during the registration period and are stored in CR's LoadTable. Rank power RP_m indicates the computing power of each individual core and is calculated using formula: $RP_m = 6 * C_m(CP) + C_m(M) + (C_m(RT)^{-2})$. In each ideal time (after waiting for new request up to a certain period of time), CR continuously computes the ranking power of cores whose status are not "BUSY". In all other situations the proposed technique calculates RP_m and stores it in LoadTable. Registration and ranking power calculation algorithm is shown in Algorithm 1.

Job Allocation

The next challenging task is distributing computation job among cores. To do this CR creates a QueueTable(Data id, Data location, Job type, Job status, Processing core ID, Processing time). Data id can be any value such as "A","B","C", etc. Data location is the location of the dataset. Job type can be either "New" or "Maintenance". Job status can be set as either "Unprocessed" or "Processed". Processing core ID is the id of the participating core. Processing time is the completion time of the assigned job. A job request may have two different operations: completely new processing (i.e., Spatial k-dominant skyline computation) or a maintenance operation. If the job request is completely new then the type field in QueueTable populate with "New" otherwise set "Maintenance". When CR receives a new job. It places the job in queue and sets job data status as "Unprocessed". In CR an independent thread runs continuously to search the queue in order to find whether or not any new job is being submitted. If a job or a set of jobs has status "Unprocessed" then the thread starts parsing other field values for each new job. Assume a new request for dataset R_p has arrived in queue. CR needs to find a suitable core to assign the computation responsibility.

If CR receives "New" request for the dataset R_p then, it searches LoadTable to find a free core say, C_m with status "FREE" to do the job. It is possible to have more than one core with "FREE" status. In such situation it selects the core C_m with highest Rank power. After that it initiates a new child thread at CR to consult the computation process with C_m. In the child thread, C_m status

Algorithm 2: Job Allocation

Create LoadTable (ID, IP Address, Port no., Computing power, Memory size, Network traffic, Rank power, Status)
QueueTable(Data id, Data location, Job type, Job status, Processing core ID, Processing time)
1. Search QueueTable for all entries with job status "Unprocessed".
2. If found then for all available dataset do the following steps
   a. Read the job type.
   b. If the job type is "New" then
      i. Search LoadTable for a core C_m with status "FREE".
      ii. If C_m found then set Status = "BUSY"
      _ Initiate new thread for assign computation:
        # Make RMI call to do the computation.
        # Receive the result.
        # Store result into a standard database.
        # Update queue, set core ID = C_m and Job status = "Processed".
        # Mark C_m Status = "FREE".
      iii. Else continue
   c. Elseif the job type is maintenance then
      i. Search QueueTable for the core C_q who processed the dataset last time.
      ii. If C_q found then
      _ Search LoadTable to get the current C_q status.
      _ If C_q Status = "FREE" then
        # Update LoadTable, set C_q Status = "BUSY".
        # Initiate the maintenance job:
          (.) Make RMI call to do the computation.
          (.) Receive result.
          (.) Store result into standard database.
          (.) Update queue, set core ID = C_m and Job status = "Processed".
          (.) Mark C_q Status = "FREE".
      Else continue
      Else continue
3. Go to step 1.

is changed from "FREE" to "BUSY". CR transfers R_p to C_m, and waits for result. Waiting period may vary depending on the size of R_m. After receiving the computation result, the resultant data is being preserved in database and C_m status is marked as "FREE". The computation time and computing core ID are also stored in CR's data storage system.

In case of "Maintenance" request CR searches QueueTable for the core say, C_q who has processed R_p most recently. After finding the core C_q, CR searches LoadTable to obtain the information on current status of C_q. If C_q is available i.e., "FREE" then CR initiates necessary steps to send the maintenance request to C_q and waits for result. After receiving the result, it updates the local database. However if C_q is not free, CR has to wait until it becomes "FREE". During the time of initiating a maintenance job to core C_q, LoadTables status field is changed from "FREE" to "BUSY". Meanwhile the dataset R_p status field in QueueTable is set to "Processed". Algorithm 2 represents the job allocation algorithm.

Populating Load and Queue Table

This section discusses about how to populate LoadTable and QueueTable. Assume two computing core (core 1 and core 2) send their registration request to CR. These requests are registered to LoadTable as shown in Table 1. Suppose at a certain period of time CR receives three data processing requests. Among the requests two for new data processing and remaining one for maintenance. At this stage the QueueTable becomes Table 2. From QueueTable CR finds some jobs with "Unprocessed" status. LoadTable shows that both cores have "FREE" status and core 1 has

Table 1: LoadTable after complete the registration task

ID	IP Add.	Port no.	Comp. Power	Mem. Size	Net. traffic	Rank Power	Status
1	192.168.0.4	2321	3.6 GHz	2048 MB	600 ms	2069.64	FREE
2	192.168.0.19	2222	2 GHz	2048 MB	300 ms	2060.0	FREE

Table 2: QueueTable after receives three jobs

Data id	Data Location	Job type	Job status	Processing core ID	Processing time
A	C:\Data\a.csv	New	Unprocessed	NA	NA
B	C:\Data\b.csv	New	Unprocessed	NA	NA
A	C:\Data\x.csv	Maintenance	Unprocessed	NA	NA

the highest ranking power. Then CR assigns the job with data id "A" to core 1 and data id "B" to core 2, and updates LoadTable and QueueTable accordingly (see Tables 3 and 4). Since now no core has "FREE" status, no new job initialization is possible until any one becomes "FREE". Suppose after 1500 seconds, computation process of data "A" has been completed. CR preserves the result. Updated LoadTable and QueueTable are shown in Tables 5 and 6, respectively. Finally, the maintenance job upon dataset "A" is assigned to core 1. Similar procedure will repeat for large number of new dataset computation as well as maintenance.

Computation Core Selection

The most challenging task is to develop a mechanism for clients end computation. In this approach, client's core, say, C_c, begins its operation by issuing a participation registration request to CR. After that C_c waits until any computation request is received. As we mentioned previously, there exist two types computation requests: "New" and "Maintenance".

If C_c receives a "New" request, it simply creates an object of computation class and initiates the spatial k-dominant skyline computation. After performing the calculation, C_c preserves resultant data and some internal status data of computing object for future maintenance operation. Preserving only resultant and status data may cause some difficulties to re-initiate k-dominate computation as well as maintenance. To overcome those problems we preserve the computing object itself instead of resultant and status data. C_c serializes the computing object, and preserves the object as Binary Logic Object Data (BLOD) into a database of file system. Then it returns the result to CR and waits for a new job request.

In case of receiving a "Maintenance" computation request for regional R_p dataset, C_c searches its local database or file system for the computation object whose dataset match with R_p. It should be found, because CR sent such request upon finding that C_c has already completed such computation in recent time. C_c de-serializes the BLOD data and constructs computation object. After performing required maintenance operation, C_c sends back the resultant data and again preserves the serialized form of computation object. Computation core selection algorithm is shown Algorithm 3.

4.2 Spatial k-dominant Skyline Computation

Assume there is a regional dataset $R_p(1 \leq p \leq x)$ as shown in Table 7. In order to prune unnecessary objects efficiently in the k-dominant skyline computation, we compute *domination power* of each

Table 3: LoadTable after assign the jobs between two cores

ID	IP Add.	Port no.	Comp. Power	Mem. Size	Net. traffic	Rank Power	Status
1	192.168.0.4	2321	3.6 GHz	2048 MB	600 ms	2069.64	BUSY
2	192.168.0.19	2222	2 GHz	2048 MB	300 ms	2060.00	BUSY

Table 4: QueueTable after assign the jobs

Data id	Data Location	Job type	Job status	Processing core ID	Processing time
A	C:\Data\a.csv	New	Processing	1	NA
B	C:\Data\b.csv	New	Processing	2	NA
A	C:\Data\x.csv	Maintenance	Unprocessed	NA	NA

Table 5: LoadTable after completing the job of core 1

ID	IP Add.	Port no.	Comp. Power	Mem. Size	Net. traffic	Rank Power	Status
1	192.168.0.4	2321	3.6 GHz	2048 MB	600 ms	2069.64	FREE
2	192.168.0.19	2222	2 GHz	2048 MB	300 ms	2060.00	BUSY

Table 6: QueueTable after completing the job of core 1

Data id	Data Location	Job type	Job status	Processing core ID	Processing time
A	C:\Data\a.csv	New	Processed	1	1500
B	C:\Data\b.csv	New	Unprocessed	2	NA
A	C:\Data\x.csv	Maintenance	Unprocessed	NA	NA

Algorithm 3: Computation Core Selection

Standard database management system (MySQL)
1. Each core with Status = "FREE" sends computation request to CR.
2. If new processing request arrives from CR then
   a. Check the job type.
   b. If the job type is "New" then
      i. Initiate new computation class object.
      ii. Perform spatial k-dominant skyline computation.
      iii. Serialize the computation object.
      iv. Preserve the computation object into standard database.
      v. Return result to the CR.
   c. Elseif the job type is "Maintenance"
   Search local database management system for processing object
      i. If found then
         # De-serialize the stored object to make it alive.
         # Perform the maintenance operation.
         # Serialize the computation object.
         # Replace the previous serialized object with the new one.
         # Return result to CR.
      ii. Else return ERROR: OBJECT NOT FOUND.
4. Return to step 2.

Table 7: Symbolic Regional Dataset, R_p

Obj.	D_1	D_2	D_3	D_4	D_5	D_6
O_1	7	3	5	4	4	3
O_2	3	4	4	5	1	3
O_3	4	3	2	3	5	4
O_4	5	3	5	4	1	2
O_5	1	4	1	1	3	4
O_6	5	3	4	5	1	5
O_7	1	2	5	3	1	2

Table 8: Ordered Domination Table

Obj.	D_1	D_2	D_3	D_4	D_5	D_6	DP	Sum	DC	IDX
O_7	$\underline{1}$	$\underline{2}$	5	3	$\underline{1}$	$\underline{2}$	4	14	3	O_5
O_5	$\underline{1}$	4	$\underline{1}$	$\underline{1}$	3	4	3	14	4	O_7
O_4	5	3	5	4	$\underline{1}$	$\underline{2}$	2	20	6	O_7
O_2	3	4	4	5	$\underline{1}$	3	1	20	5	O_7
O_6	5	3	4	5	$\underline{1}$	5	1	23	5	O_7
O_3	4	3	2	3	5	4	0	21	5	O_7
O_1	7	3	5	4	4	3	0	26	6	O_7

object. We sort objects in descending order by domination power. If more than one objects have the same domination power then sort those objects in ascending order of the sum value. This order reflects how to k-dominate other objects.

Table 8 is the sorted object sequence of Table 7, in which the column "DP" is the domination power and the column "Sum" is the sum of all values. In the sequence, object O_7 has the highest domination power 4. Note that object O_7 dominates all objects below it in four attributes, D_1, D_2, D_5, and D_6.

After computing the sorted object sequence, we compute dominated counter (DC) and dominant index (IDX). Detailed algorithm can be found in [15]. The dominated counter (DC) indicates the maximum number of dominated dimensions by another object in R_p. The dominant index (IDX) is the strongest dominator. That means IDX keeps the record of the corresponding strongest dominator for each object.

The columns DC and IDX of Table 8 show the result of the procedure. $Sky_k(R_p)$ is a set of objects whose DC is less than k. According to the dominated counter, we can see that $Sky_6(R_p) = \{O_7, O_5, O_2, O_6, O_3\}$, $Sky_5(R_p) = \{O_7, O_5\}$, and $Sky_4(R_p) = \{O_7\}$. Since there is no object whose DC value is less than 3, thus $Sky_3(R_p) = \{\emptyset\}$.

4.3 Spatial k-dominant Skyline Maintenance

In this section, we discuss the maintenance problem of $Sky_k(R_p)$ after an update has occurred in R_p. In the maintenance phase, we keep a vector that contains the minimal value for each dimension, which we call the minimal vector. The minimal vector of Table 8 is $\{1, 2, 1, 1, 1, 2\}$. We also keep an inverted index of the dominant index column (IDX). Table 9 is the inverted index of Table 8.

Lemma 1: Assume $O_1.D_s \leq O_2.D_s$ for all dimensions D_s $(s = 1, \cdots, n)$ for $O_1, O_2 \in R_p$. If O_1 is not deleted from R_p, O_2 will never be in the k-dominant skyline of R_p.

Proof: Since O_2 is n-dominated by O_1, it is also k-dominated by O_1 because $(k \leq n)$. Therefore, while O_2 is in the R_p, there will be at least one object, namely O_1, that k-dominates it and consequently cannot become a k-dominant skyline. It implies that only $Sky_n(R_p)$ objects are relevant for the k-dominant skyline maintenance task. $\diamondsuit$

Table 9: Inverted Index

Obj.	dominated
O_5	O_7
O_7	O_5, O_4, O_2, O_6, O_3, O_1

Table 10: Ordered Domination Table after Insertions

Obj.	DP	Sum	DC	IDX
O_7	4	14	3	O_5
O_5	3	14	4	O_7
O_4	2	20	6	O_7
O_2	1	20	5	O_7
O_6	1	23	5	O_7
O_9	0	18	5	O_2
O_3	0	21	5	O_7
O_1	0	26	6	O_7
O_8	0	36	6	O_7

Obj.	DP	Sum	DC	IDX
O_7	4	14	3	O_5
O_5	3	14	4	O_7
O_{10}	2	13	4	O_7
O_4	2	20	6	O_7
O_2	1	20	5	O_7
O_6	1	23	5	O_7
O_9	0	18	6	O_{10}
O_3	0	21	5	O_7
O_1	0	26	6	O_7
O_8	0	36	6	O_7

Insertion

Assume O_I is inserted into R_p. We maintain $Sky_k(R_p)$ by examining the dominated counter (DC) by scanning the sorted object sequence. If the DC value of an object is updated, we also update the inverted index. During the update procedure, if O_I is n-dominated by another object, then we can stop the procedure immediately.

After updating DC and IDX, we insert O_I into the sorted object sequence. We use a skip list data structure for the sequence so that we can insert efficiently.

Assume we insert $O_8 = (6, 6, 6, 6, 6, 6)$ in the running example. By comparing with the first object O_7, we can find that O_8 is 6 dominated by O_7. Therefore, we immediately complete the update of DC and IDX. Then, we insert O_8 into the last position of the sorted sequence. Next, we insert $O_9 = (3, 4, 4, 2, 2, 3)$. O_9 is not 6 dominated by any object and we can find that the strongest dominator of O_9 is O_2. Then, DC and IDX are updated as in Table 10 after inserting O_9. Next, we insert $O_{10} = (2, 2, 3, 1, 2, 3)$. O_{10} is not 6-dominated by any object and the strongest dominator of O_{10} is O_7. Moreover, O_{10} becomes the strongest dominator of O_9. Then, DC and IDX are updated as in Table 10 after inserting O_{10}.

Deletion

Assume O_D is deleted from R_p. We check the inverted index to examine whether there is O_D's record in the index.

Note that in Table 9 "Obj." column in each record is the strongest dominator for objects in the "dominated" column. Therefore, if there is no O_D's record in the inverted index, we do not have to change the dominated counter (DC) for other objects. Otherwise, we initialize DC for each object in the O_D's index record. Then, we perform the pairwise comparison. In this case, we do not have to examine DC for objects that are not in the O_D's index record and therefore it is not costly.

Consider the running example again. Assume we delete O_{10}. We examine DC of O_9 by scanning Table 10. The updated result is Table 10.

Update

In this section, we propose maintenance solution for update operation. Although one can handle update operation with consecutive deletion and insertion. However, single update operation is usually faster than consecutive delete and insert operation. Assume O_U is updated from R_p. Then,

Table 11: Ordered Domination Table after Updates

Obj.	DP	Sum	DC	IDX
O_7	4	14	3	O_5
O_5	3	14	4	O_7
O_4	2	20	6	O_7
O_3	1	17	5	O_7
O_2	1	20	5	O_7
O_6	1	23	6	O_3
O_1	0	26	6	O_7

Obj.	DP	Sum	DC	IDX
O_5	3	14	3	O_4
O_4	2	20	4	O_7
O_7	2	24	4	O_5
O_2	1	20	4	O_5
O_6	1	23	5	O_4
O_3	0	21	5	O_5
O_1	0	26	6	O_4

Table 12: Response time for heterogeneous cores

Core Type	Time (ms)
Single Core	603455
Core2	362701
Core i3	106986
Core i5	61329
Core i7	30458

same as delete operation, we have to check the inverted index to examine whether there is O_U's record in the index. If there is no O_U's record in the inverted index, then we need one scan to revise the dominated counter (DC) for updated object as well as all other data objects. Otherwise, we initialize DC for each object from scratch. However, in this situation we argue that compared with non-IDX objects, the number of IDX objects is very few and therefore update operation is also not very costly.

Consider Table 8 and select a non-IDX object such as O_3 for update. Assume we update D_5 value of this object from 5 to 1. Then after dominance checking with other objects we find that O_3 becomes the strongest dominator of object O_6. This update procedure is shown in Table 11. Again from Table 8 if we select an IDX object such as O_7. In this case, we update the values of D_1 and D_5 from 1 to 6. The revised result after this update is shown in Table 11.

5 Performance Evaluation

We conduct a series of experiments to evaluate the effectiveness and efficiency of our proposed methods. There are no other existing techniques that deal with the problem of maintaining k-dominant skyline. So in this paper, we compare our methods against $MSKS$, which was proposed in [16]. Our heterogeneous simulation takes five different cores (PCs) that are connected to a coordinator (server). Processors of those cores are as follows: single core, core2, core i3, core i5, and core i7. They have 3GB main memory. We choose independent distribution with 6 dimensionality and 100k data cardinality to evaluate each core performance. Table 12 shows the computation time of each core. We observe that high power computing cores take less time than those of low power cores. This result establishes the necessity of rank power computation.

Next, our homogeneous simulation takes similar type of cores. The number of cores is moderately small (from 2 to 8). Each of the cores has an Intel(R) Core2 Duo 2 GHz CPU and 3GB main memory. We evaluate our algorithms against $MSKS$ using different types of datasets. As benchmark, we use the datasets proposed by Borzsony et al. [2], in which there are three types of synthetic data distributions: 'correlated', 'anti-correlated', and 'independent'. The generation of the synthetic datasets is controlled by three parameters, n, "Size", and "Dist", where n is the number of attributes, "Size" is the total number of objects in the dataset, and "Dist" can be any of the three distributions. In addition, we have generated smaller synthetic datasets for all insertion experiments. For example to conduct insertion experiment on 100k synthetic dataset, we have also generated additional 10k dataset. As for delete and update operations, we choose random object from the experimental dataset.

5.1 Spatial k-dominant Computation Varying Core Number

We first study the effect of spatial k-dominant computation varying core number on our techniques. We use anti-correlated, independent, and correlated datasets with dimensionality n to 7, cardinality to 100k, and k to 6. The number of cores varies from 2 to 8. Figure 3(a), (b), and (c) shows the time to compute spatial k-dominant skyline. Figure 3 shows that the performance of proposed methods outperform than $MSKS$ for all data distribution.

5.2 Effect of Data Distribution

To study the effect of data distributions we use anti-correlated, independent, and correlated datasets with dimensionality n to 7, cardinality to 100k, k to 6, and core number to 8. figure 4(a), (b), and (c) shows the time to maintain spatial k-dominant skyline for the maintenance ranges from 1% to 5%. In this experiment, 1% maintenance for 100k dataset implies that there are 333 insertions, 333 deletions, and 334 updates (total 1k updates) occurred in the dataset. Figure 4 shows that the performance of both methods deteriorates significantly with the increase of maintenance size. We further notice that for anti-correlated dataset, many spatial k-dominant skyline objects retrieved as a result the maintenance cost of this distribution incurs high computational overheads.

5.3 Effect of Dimensionality

For this experiment, we choose the anti-correlated datasets and the core number is 8. We set the data cardinality to 100k and vary dataset dimensionality n ranges from 4 to 8 and k from 3 to 7. Figure 5(a), (b), and (c) shows the update performance. The $MSKS$ technique is highly affected by the curse of dimensionality, i.e., as the space becomes sparser its response time rapidly decreases. Figure 5 shows that if the dimensionality and the maintenance ratio increase the response time of proposed methods grows steadily, which is much less than that of $MSKS$.

5.4 Effect of Cardinality

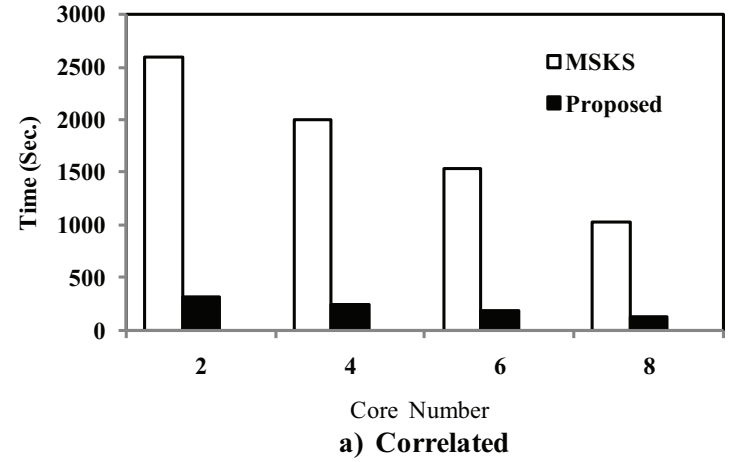
For this experiment, we select the anti-correlated datasets with varying dataset cardinality ranges from 100k to 200k, set the value of n to 7, k to 6, and core number to 8. Figure 6(a), (b), and (c) shows the time to maintain k-dominant skyline for maintenance ranges from 1% to 5%. The result shows that if the maintenance ratio and data cardinality increase the response time of proposed methods also increases. However, the response time is much smaller than that of $MSKS$.

6 Conclusion

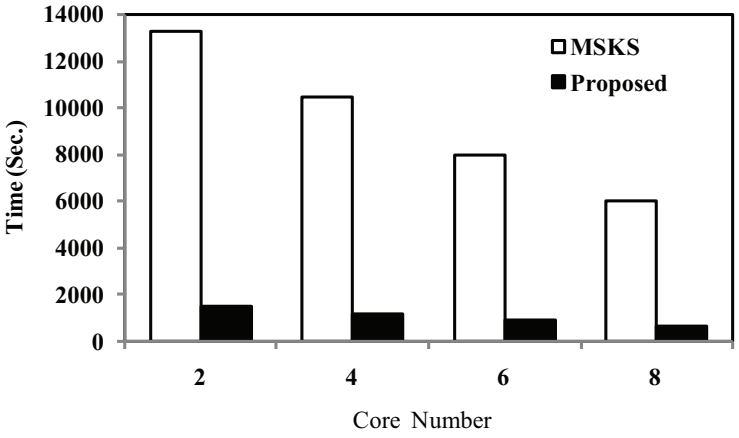
k-dominant skyline has side effect of taking larger time for centralized computation. Centralized system also has maintenance problem. To solve those problems, we consider regionally distributed k-dominant skyline and present an efficient architecture for computing the query result as well as the maintenance problem. We introduced registration and ranking power calculation, job allocation, and computation core selection algorithms for efficient load distribution and shortening the response time. Our algorithms consistently outperform against $MSKS$ algorithm. Intensive experiments show the efficiency and scalability of our proposed methods. However, current architecture can handle small number of cores. We leave the work to support large number of cores for future research.

References

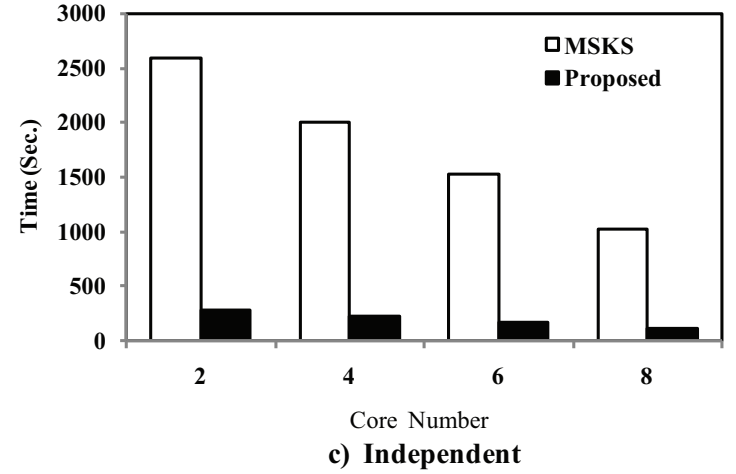
[1] T. Xia, D. Zhang, and Y. Tao, "On Skylining with Flexible Dominance Relation", in Proceedings of ICDE, 2008, pp. 1397-1399.



a) Correlated

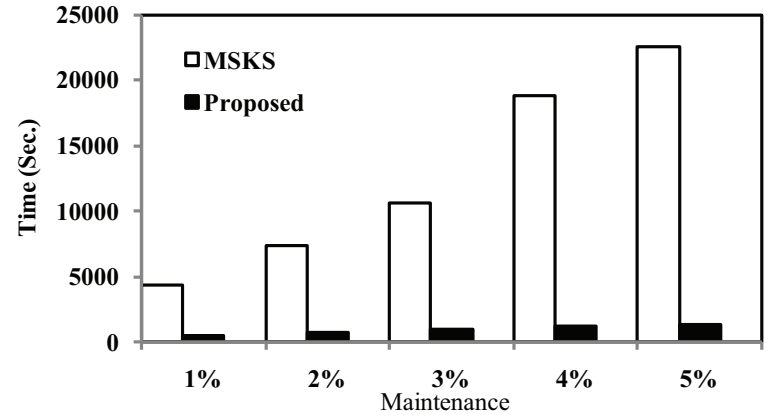


b) Anti-correlated

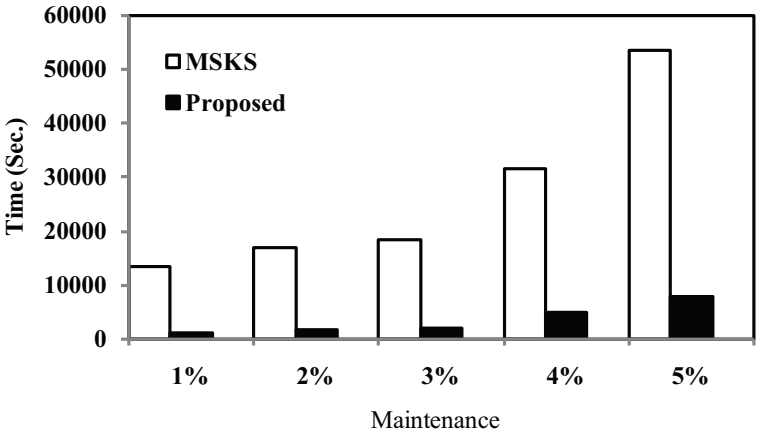


c) Independent

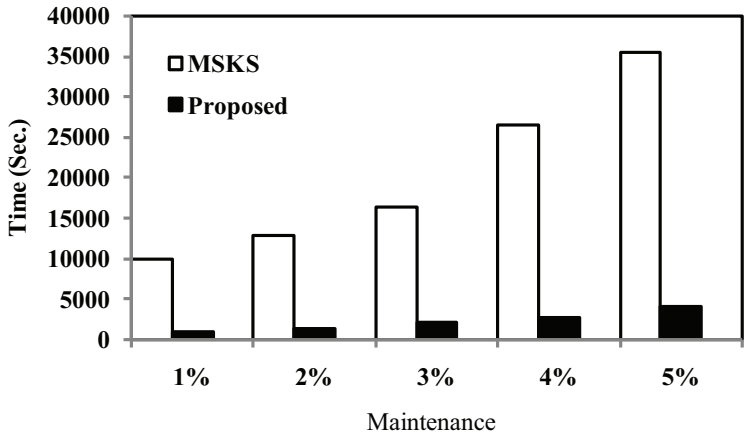
Figure 3: Computation performance for different core number



a) Correlated

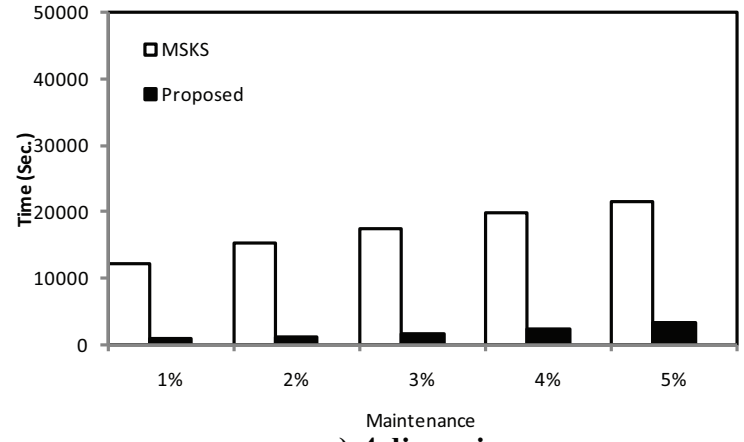


b) Anti-correlated

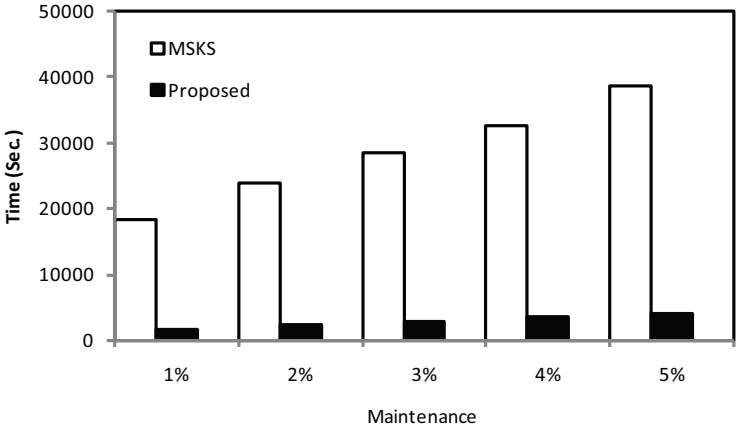


c) Independent

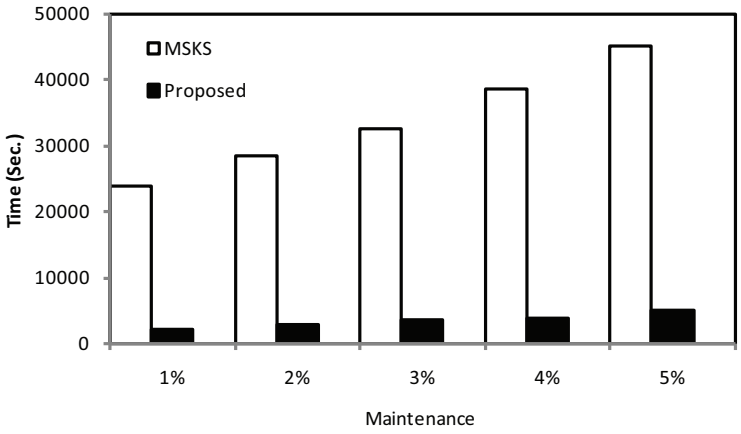
Figure 4: Maintenance performance for different data distributions



a) 4 dimension



b) 6 dimension



c) 8 dimension

Figure 5: Maintenance performance for different dimensions

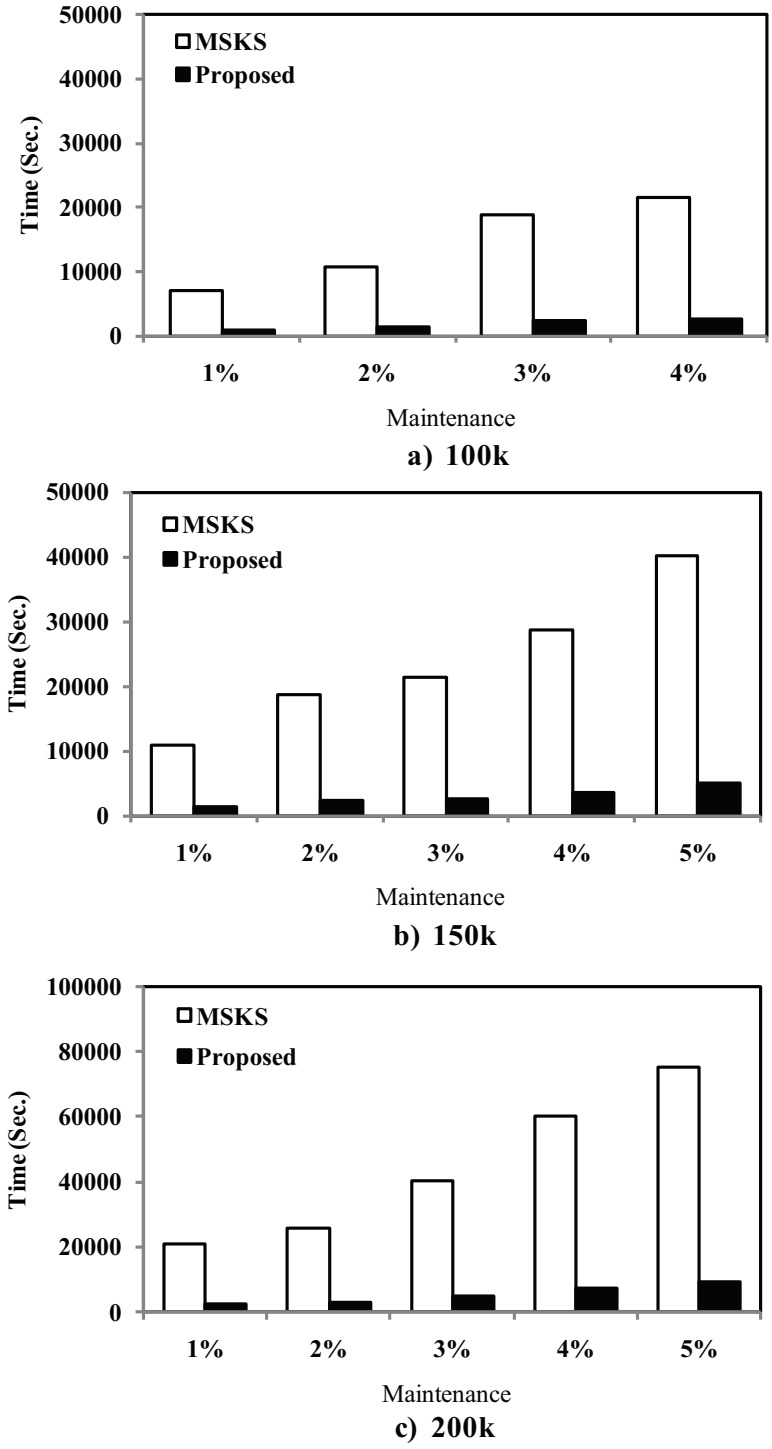


Figure 6: Maintenance performance for different data cardinality

[2] S. Borzsonyi, D. Kossmann, and K. Stocker, "The skyline operator", in Proceedings of ICDE, 2001, pp. 421-430.

[3] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: An online algorithm for skyline queries", in Proceedings of VLDB, 2002, pp. 275-286.

[4] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with Presorting", in Proceedings of ICDE, 2003, pp. 717-719.

[5] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems", in ACM Transactions on Database Systems, vol. 30(1), pp. 41-82, March 2005.

[6] C. Y. Chan, H. V. Jagadish, K-L. Tan, A-K. H. Tung, and Z. Zhang, "Finding k-Dominant Skyline in High Dimensional Space", in Proceedings of ACM SIGMOD, 2006, pp. 503-514.

[7] K.-L. Tan, P.-K. Eng, and B. C. Ooi, "Efficient Progressive Skyline Computation", in Proceedings of VLDB, 2001, pp. 301-310.

[8] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware", in Proceedings of ACM PODS, 2001, pp. 102-113.

[9] A. Vlachou, C. Doulkeridis, Y. Kotidis, and M. Vazirgiannis, "SKYPEER: Efficient Subspace Skyline Computation over Distributed Data", in Proceedings of ICDE, 2007, pp. 416-425.

[10] K. Fotiadou and E. Pitoura, "BITPEER: Continuous Subspace Skyline Computation with Distributed Bitmap Indexes", in Proceedings of DaAMaP, 2008, pp. 35-42.

[11] C. Y. Chan, H. V. Jagadish, K-L. Tan, A-K. H. Tung, and Z. Zhang, "On High Dimensional Skylines", in Proceedings of EDBT, 2006, pp. 478-495.

[12] Y. Tao, X. Xiao, and J. Pei, "Subsky: Efficient Computation of Skylines in Subspaces", in Proceedings of ICDE, 2006, pp. 65-65.

[13] W.-T Balke, U. Guntzer, and J. X. Zheng, "Efficient Distributed Skylining for Web Information Systems", in Proceedings of EDBT, 2004, pp. 256-273.

[14] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi, "Skyline Queries Against Mobile Lightweight Devices in Manets", in Proceedings of ICDE, 2006, pp. 66.

[15] M. A. Siddique and Y. Morimoto, "Efficient Maintenance of all k-Dominant Skyline Query Results for Frequently Updated Database", in The International Journal on Advances in Software, Vol. 3, No. 3 & 4, 2010, pp. 424-433.

[16] M. A. Siddique, A. Zaman, M. M. Islam, and Y. Morimoto, "Multicore based Spatial k-dominant Skyline Computation", in Proceedings of the 3rd Int'l Conference on Networking and Computing (ICNC), 2012, pp. 188-194.

[17] W.-T. Balke, U. Guntzer, and J. X. Zheng, "Efficient distributed skylining for web information systems", in Proceedings of EDBT, 2004, pp. 256-273.

[18] L. Chen, B. Cui, H. Lu, L. Xu, and Q. Xu, "iSky: Efficient and progressive skyline computing in a structured P2P network", in Proceedings of ICDCS, 2008, pp. 160-167.

[19] B. Cui, H. Lu, Q. Xu, L. Chen, Y. Dai, and Y. Zhou, "Parallel distributed processing of constrained skyline queries by filtering", in Proceedings of ICDE, 2008, pp. 546-555.

[20] K. Fotiadou and E. Pitoura, "BITPEER: Continuous subspace skyline computation with distributed bitmap indexes", in Proceedings of DAMAP, 2008, pp. 35-42.

[21] K. Hose, C. Lemke, and K.-U. Sattler, "Processing relaxed skylines in PDMS using distributed data summaries", in Proceedings of CIKM, 2006, pp. 425-434.

[22] J. B. Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nrvag "AGiDS: A grid-based strategy for distributed skyline query processing", in Proceedings of GLOBE, 2009.

[23] A. Vlachou, C. Doulkeridis, and Y. Kotidis "Angle-based space partitioning for efficient parallel skyline computation", in Proceedings of SIGMOD, 2008, pp. 227-238.

[24] A. Vlachou, C. Doulkeridis, Y. Kotidis, and M. Vazirgiannis, "SKYPEER: Efficient subspace skyline computation over distributed data", in Proceedings of ICDE, 2007, pp. 416-425.

[25] A. Vlachou, C. Doulkeridis, Y. Kotidis, and M. Vazirgiannis, "Efficient routing of subspace skyline queries over highly distributed data", Trans. on Knowledge and Data Engineering (TKDE), Vol. 22, No. 12, 2010, pp. 1694-1708.

[26] A. Vlachou and K. Nrvag, "Bandwidth-constrained distributed skyline computation", in Proceedings of MobiDE, 2009.

[27] S. Wang, B. C. Ooi, A. K. H. Tung, and L. Xu, "Efficient skyline query processing on peer-to-peer networks", in Proceedings of ICDE, 2007, pp. 1126-1135.

[28] S. Wang, Q. H. Vu, B. C. Ooi, A. K. H. Tung, and L. Xu, "Skyframe: A framework for skyline query processing in peer-to-peer systems", VLDB Journal, Vol. 18, No. 1, 2009, pp. 345-362.

[29] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi, "Parallelizing skyline queries for scalable distribution", in Proceedings of EDBT, 2006, pp. 112-130.