Efficient Skyline Computation on Big Data

Xixian Han, Jianzhong Li, *Member*, *IEEE*, Donghua Yang, and Jinbao Wang

Abstract—Skyline is an important operation in many applications to return a set of interesting points from a potentially huge data space. Given a table, the operation finds all tuples that are not dominated by any other tuples. It is found that the existing algorithms cannot process skyline on big data efficiently. This paper presents a novel skyline algorithm SSPL on big data. SSPL utilizes sorted positional index lists which require low space overhead to reduce I/O cost significantly. The sorted positional index list L_j is constructed for each attribute A_j and is arranged in ascending order of A_j. SSPL consists of two phases. In phase 1, SSPL computes scan depth of the involved sorted positional index lists. During retrieving the lists in a round-robin fashion, SSPL performs pruning on any candidate positional index to discard the candidate whose corresponding tuple is not skyline result. Phase 1 ends when there is a candidate positional index seen in all of the involved lists. In phase 2, SSPL exploits the obtained candidate positional indexes to get skyline results by a selective and sequential scan on the table. The experimental results on synthetic and real data sets show that SSPL has a significant advantage over the existing skyline algorithms.

Index Terms—Big data, skyline, pruning, SSPL

1 Introduction

As a consequence of Moore's law, circuit densities increase fourfold every three years [20]. Over the last two decades, disk capacity has improved 75,000 times from 40 MB (Maxtor 7000 Series IDE 3524) in 1991 to 3TB (Seagate Barracuda XT Series ST33000651AS) in 2012, while disk transfer rate has improved only 750× from 800 KB/s to 600 MB/s [27], [32]. The ratio of disk capacity to disk transfer rate increases by 10× per decade. Therefore, disk is becoming slower from the view of applications because of much higher data volume that they need to store and process.

Skyline is an important operation in many applications to return a set of interesting points from a potentially huge data space [7], [23]. A subset of attributes is designated as skyline criteria, on which the dominance relationship between tuples is defined. Given two tuples p and q in a table, p dominates q if, among skyline criteria, p is not larger than q in all attributes and strictly smaller than q in at least one attribute. Skyline finds all tuples that are not dominated by any other tuples.

Recently, skyline has attracted extensive attention and many algorithms are proposed. A set of skyline algorithms, such as Bitmap [30], NN [22], BBS [26], SUBSKY [31], and ZBtree [24], utilize indexes to reduce the explored data space and return skyline results. However, because of the prohibitive pre-computation cost and space overhead to cover the attributes involved in skyline on big data, index-based algorithms have serious limitations and the used indexes can only be built on a small and selective set of attribute combinations. A set of more scalable and practical skyline algorithms, such as divide and conquer [7], [23], [28], BNL [7], SFS [12], and LESS [19], do not require preprocessing steps or data-structures. But they have to scan the entire table at least once, which will incur high I/O cost on big data. Nowadays, big data is commonly stored and used in scientific research and business application. For example, Walmart maintains a 4PB data warehouse for retail information with dozens of attributes [8], and in experiments with the high-speed collision of subatomic particles, 300TB of data per year is generated with up to 200 attributes [17]. It is well known that scanning big data is rather expensive and it will take several hours to scan 1TB of data even on a high-end computer. In a word, the existing algorithms cannot process skyline on big data efficiently.

People usually expect to get results quickly, and they will become impatient to wait several hours for results. In view of the problems of the existing algorithms, this paper proposes a novel skyline algorithm on big data, skyline with sorted positional index lists (SSPL), to return skyline results efficiently. The algorithm utilizes the preconstructed data-structures which require low space overhead to reduce I/O cost significantly. Given a table $T(A_1, A_2, \ldots, A_M)$, SSPL pre-builds a sorted positional index list $L_j (1 \leq j \leq M)$ for each attribute A_j, and L_j is arranged in ascending order of A_j. SSPL consists of two phases: obtaining the candidate positional indexes (phase 1) and retrieving the skyline results (phase 2). In phase 1, SSPL first retrieves the sorted positional index lists $\{L_1, L_2, \ldots, L_m\}$ involved by skyline criteria $\{A_1, A_2, \ldots, A_m\}$ in a round-robin fashion. A mathematical analysis is proposed to compute scan depth d of the lists in phase 1. It is guaranteed that the candidate positional indexes corresponding to the skyline results are contained in the first d elements in $\{L_1, L_2, \ldots, L_m\}$. In phase 1, SSPL

- *X. Han, J. Li, and J. Wang are with the School of Computer Science and Technology, Harbin Institute of Technology, PO Box 750, Harbin, Heilongjiang, China 150001. E-mail: xxhan1981@163.com, lijzh@hit.edu.cn, wangjinbaosky@gmail.com.*
- *D. Yang is with the Academy of Fundamental and Interdisciplinary Sciences, Harbin Institute of Technology, PO Box 750, Harbin, Heilongjiang, China 150001. E-mail: yang.dh@hit.edu.cn.*

Manuscript received 22 Mar. 2012; revised 13 July 2012; accepted 10 Oct. 2012; published online 15 Oct. 2012.
Recommended for acceptance by W.-S. Han.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2012-03-0191.
Digital Object Identifier no. 10.1109/TKDE.2012.203.

performs pruning on any candidate positional index retrieved from $\{L_1, L_2, \ldots, L_m\}$ to discard the candidate whose corresponding tuple is not skyline result. This paper proposes general rules and mathematical analysis for pruning operation. Phase 1 ends when there is a candidate positional index seen in all lists of $\{L_1, L_2, \ldots, L_m\}$. In phase 2, SSPL exploits the obtained candidate positional indexes to compute skyline results by a selective and sequential scan on the table. At first glance, the sorted positional index lists for SSPL are similar to the sorted column files in [9] and [14]. However, the most significant idea for SSPL is its pruning operation. Unlike the sorted column files which are used to support sorted retrieval mainly, the sorted positional index lists are the data-structures to facilitate pruning and reduce the candidate tuples significantly. Although SSPL is an approximate method to obtain skyline results, its probability of correctness is extremely high. The extensive experiments are conducted on two sets of terabyte synthetic data and a set of gigabyte real data, and the experimental results show that compared to the existing algorithms, SSPL involves up to six orders of magnitude fewer tuples, and obtains up to three orders of magnitude speedup.

The main contributions of this paper are listed as follows:

- This paper presents a novel skyline algorithm SSPL on big data, which can utilize some small preconstructed data-structures to reduce I/O cost significantly.
- The behavior analysis of SSPL is proposed to determine scan depth of the sorted positional index lists.
- This paper devises pruning operation on the candidate positional indexes, and the mathematical analysis for pruning is presented in this paper.
- The experimental results show that SSPL has a significant advantage over the existing skyline algorithms.

The remainder of the paper is organized as follows: Section 2 provides background and problem definition. Section 3 reviews related work. Section 4 describes the SSPL algorithm. Performance evaluation is shown in Section 5. We conclude this paper in Section 6.

2 Background and Problem Definition

Given a table $T(A_1, A_2, \ldots, A_M)$, $\forall t \in T$, let us denote by $t[j]$ the jth attribute A_j of t. Without loss of generality, let a subset of attributes $AS_{skyline} = \{A_1, A_2, \ldots, A_m\}$ be skyline criteria, and the dominance relationship between tuples is defined on $AS_{skyline}$. $\forall t_1, t_2 \in T$, t_1 dominates t_2 (denoted by $t_1 \succ t_2$), if $\forall j (1 \le j \le m)$, $t_1[j] \le t_2[j]$, and $\exists j (1 \le j \le m)$, $t_1[j] < t_2[j]$, i.e., $t_1 \succ t_2 \Leftrightarrow (\bigcap_{j=1}^{m}(t_1[j] \le t_2[j])) \cap (\bigcup_{j=1}^{m}(t_1[j] < t_2[j]))$. For clarity, we assume that *min* condition only is used for skyline computation. However, the algorithm here can be extended to process any combination of conditions (*min* or *max*).

Skyline query. Given a table T, skyline query returns a subset $SKY(T)$ of T, in which $\forall t_1 \in SKY(T)$, $\nexists t_2 \in T$, $t_2 \succ t_1$.

Given tuple number n in table T and size m of skyline criteria, the expected number s of skyline results under component independence is known [3], [18]: $s = H_{m-1,n}$, here $H_{m,n}$ is the mth order harmonic of n. For any $n > 0$, $H_{0,n} = 1$. For any $m > 0$, $H_{m,0} = 0$. For any $n > 0$ and $m > 0$, $H_{m,n}$ is inductively defined as: $H_{m,n} = \sum_{i=1}^{n} \frac{H_{m-1,i}}{i}$, $H_{m,n}$ can be approximated as:

$$H_{m,n} \approx \frac{H_{1,n}^{m}}{m!} \approx \frac{(\ln n + \gamma)^m}{m!},$$

where $\gamma = \lim_{n\to\infty}(H_{1,n} - \ln n)$.

According to the computation formula of $H_{m,n}$, it is found that the number of skyline results does not change significantly as the tuple number increases, while it is very sensitive to the size of skyline criteria. For example, given $m = 3$, when n increases from 10^5 to 10^9, s changes from 66 to 214. Given $n = 10^9$, when m increases from 2 to 5, s changes from 20 to 7,684. Although the absolute number of skyline results is large, its proportion among all tuples is rather small. For example, given $m = 5$ and $n = 10^9$, $\frac{s}{n} = 7.684 \times 10^{-6}$.

3 Related Work

3.1 Index-Based Algorithms

Index-based skyline algorithms utilize the preconstructed data-structures to avoid scanning the entire data set.

Tan et al. [30] make use of bitmap to compute skyline of a table $T(A_1, A_2, \ldots, A_d)$. Given a tuple $x = (x_1, x_2, \ldots, x_d) \in T$, x is encoded as a b-bit bit-vector, $b = \sum_{i=1}^{d} k_i$ (k_i is the cardinality of A_i). We assume that x_i is the (j_i)th smallest value in A_i, the k_i-bit bit-vector representing x_i is set as follows: bit 1 to bit $j_i - 1$ are set to 0, bit j_i to bit k_i are set to 1. The encoded table is stored as bit-transposed files [35], let BS_{ij} represent the bit file corresponding to the jth bit in the ith attribute A_i. It is given that a tuple $x = (x_1, x_2, \ldots, x_d) \in T$ and x_i is the (j_i)th smallest value in A_i. Let $A = BS_{1j_1} \& BS_{2j_2} \& \ldots \& BS_{dj_d}$ where $\&$ represents the bitwise *and* operation. And let $B = BS_{1(j_1-1)} | BS_{2(j_2-1)} | \ldots | BS_{d(j_d-1)}$ where $|$ represents the bitwise *or* operation. If there is more than a single one-bit in $C = A \& B$, x is not a skyline tuple. Otherwise, x is a skyline tuple.

Kossmann et al. [22] propose NN algorithm to process skyline query. NN utilizes the existing methods for nearest neighbor search to split data space recursively. By a preconstructed R-tree, NN first finds the nearest neighbor to the beginning of the axes. Certainly, the nearest neighbor is a skyline tuple. Next, the data space is partitioned by the nearest neighbor to several subspaces. The subspaces that are not dominated by the nearest neighbor are inserted into a *to-do* list. While the *to-do* list is not empty, NN removes one of the subspaces to perform the same process recursively. During the space partitioning, overlapping of the subspaces will incur duplicates, NN exploits the methods: Laisser-faire, Propagate, Merge and Fine-grained Partitioning, to eliminate duplicates.

Papadias et al. [26] develop a progressive algorithm BBS based on nearest neighbor search. BBS applies branch-and-bound strategy. It begins with root node of R-tree, and inserts all its child-nodes in a min-heap according to their

distances to the beginning of the axes. Next, the node with the minimum distance is selected to expand the heap, i.e., remove the node and insert all its child nodes. Each node needs to be checked for dominance twice: before it is inserted into the heap and before it is expanded. If the node is dominated by the current skyline results, it is discarded. If the node selected to expand is data point, it is a part of skyline results. BBS only performs a single access to the nodes of R-tree that may contain skyline results and does not retrieve duplicates.

Tao et al. [31] propose a technique SUBSKY to perform skyline in subspaces. SUBSKY converts each d-dimensional point p to a 1D value $f(p)$ by a specified distance function: $f(p) = max_{i=1}^{d}(1 - p[i])$ (The coordinates of p are within [0, 1]). The points are accessed in descending order of their $f(p)$ values by a B-tree. Meanwhile, SUBSKY maintains the current skyline set S_{sky} and the largest $min_{i=1}^{d}(1 - p[i])$ value U of the points in S_{sky}. SUBSKY terminates when U is larger than the $f(p)$ value of the next point retrieved by B-tree. Besides, SUBSKY is extended to process clustered data by multiple anchors to achieve a better performance.

Lee et al. [24] develop a suite of skyline algorithms which scale well to dimensionality and cardinality. Z-curve is adopted here to map data points in a multidimensional space to a 1D space, and each point is represented by Z-address. A new index ZBtree is proposed to organize data points in accordance with monotonic Z-address. ZBtree divides Z-order curve into disjoint segments with each of which representing a region. ZSearch is presented to process skyline efficiently based on ZBtree. It traverses the ZBtree to visit the regions and potential skyline points in a depth-first order. And a stack is used to keep the unexplored paths. At each round, the region of a node popped from the stack is examined against the current skyline results. If the region is not dominated, the node is further explored. The data points of the leaf node which is not dominated are compared with the current skyline results. ZSearch terminates when the stack is empty.

To sum up, because of the prohibitive precomputation cost and space overhead, index-based algorithms have serious limitations. It is much expensive for bitmap algorithm to perform preconstruction and computation of the skyline results. The bit-vector length of each encoded tuple in bitmap algorithm equals the sum of cardinalities of all attributes. If some attributes have high cardinalities, the space overhead for storage is large. Besides, for checking whether each tuple in table is a part of skyline, bitmap algorithm has to retrieve the corresponding bit-transposed files involving all tuples. For tree-based algorithms, although the size of skyline criteria is typically small, the combination of the attributes over which the queries are posed can be quite large. Given a table with M attributes and skyline criteria involves not more than m attributes, tree-based algorithms have to build $\sum_{i=2}^{m}\binom{M}{i}$ indexes to cover the attributes used in queries. That is, the number of indexes has an exponential relationship with the number of attributes. Therefore, due to prohibitive computation cost and space overhead, index-based algorithms are not suitable for processing skyline on big data.

3.2 Generic Algorithm

Generic skyline algorithms do not require preprocessing steps or data-structures, they are performed on the table directly.

The computation of skyline results is also called maximal vector problem. Kung et al. [23] propose a basic divide-and-conquer algorithm DD&C to process maximal vector problem, together with its theoretical analysis. DD&C splits the input into two partitions P_1 and P_2 by the median along certain attribute d_p. $\forall t_1 \in P_1, \forall t_2 \in P_2$, we have $t_1.d_p \leq t_2.d_p$. DD&C is recursively executed on P_1 and P_2 along the remaining attributes to get maximal vector results S_1 and S_2, respectively. Then DD&C merges S_1 and S_2 to return the results by discarding the dominated vectors in S_2.

LD&C [3] and FLET [4] improve DD&C. LD&C first does a basic divide-and-conquer recursion, randomly splitting the input into two partitions. If the generated partition is below the threshold in size, LD&C directly computes the skyline results of the partition. At last, LD&C invokes DD&C to merge the skyline results of partitions. FLET first determines a virtual tuple t_1 before execution. During scanning the input, any tuple dominated by t_1 is discarded directly. If there occurs a tuple t_2 that dominates t_1, t_1 is replaced by t_2 and after scanning the input, FLET applies DD&C to compute the skyline results of the remaining tuples. Otherwise, if there is no such tuple t_2, FLET invokes DD&C to compute skyline on the entire input.

Börzsönyi et al. [7] devise an external divide-and-conquer algorithm SD&C to deal with skyline query on large input. SD&C splits the input into several partitions $P_1, P_2, \ldots, P_k$ and guarantees that each partition $P_i (1 \leq i \leq k)$ can fit into memory. Then the existing memory skyline algorithms are applied to obtain skyline results S_i of each partition P_i. To return the required results, SD&C merges $S_i (1 \leq i \leq k)$ pairwise by a bushy merge tree. Here, when merging S_i and S_j $(1 \leq i \neq j \leq k)$, the tuples from S_i and S_j are compared with each other to discard the dominated tuples.

Sheng and Tao [28] propose a novel divide-and-conquer external skyline algorithm with a lower I/O complexity. The algorithm is on the basis of the processing on 3-d space as below. The algorithm assumes that the input is arranged in ascending order of x-coordinate. At first, the algorithm splits the input by the disjoint intervals in x-coordinate into $k = \Theta(M/B)$ partitions $P_1, P_2, \ldots, P_k$ with roughly the same size such that each point in $P(i)$ has a smaller x-coordinate than all points in $P(j)$ for any $1 \leq i < j \leq k$ (M is the capacity of main memory and B is the capacity of block). The memory-resident skyline algorithms are invoked to obtain skyline results $\Sigma(1), \Sigma(2), \ldots, \Sigma(k)$ of $P_1, P_2, \ldots, P_k$, respectively, if they fit into memory. Otherwise, if $P_i (1 \leq i \leq k)$ does not fit in memory, it is further divided into smaller partitions, each of which is processed recursively. Before $\Sigma(i) (1 \leq i \leq k)$ is outputted to disk, the algorithm guarantees that $\Sigma(i) (1 \leq i \leq k)$ is arranged in ascending order of y-coordinate. Then, according to y-coordinate, the algorithm performs a k-way merging on $\Sigma(i) (1 \leq i \leq k)$. During the k-way merging, the algorithm maintains by $\lambda(i)$ the minimum z-coordinate of all the tuples seen already from $\Sigma(i)$. For each candidate tuple

$p \in \Sigma(i)$ returned in k-way merging, p is a skyline result if its z-coordinate is less than $\lambda(j)(\forall j < i)$.

Börzsönyi et al. [7] develop BNL algorithm, which sequentially scans table and maintains a set of candidate skyline results in memory buffer. For each tuple t seen during scanning, three cases are considered: 1) t is dominated by a candidate tuple in memory buffer, t is discarded directly; 2) t dominates some candidate tuples in memory buffer, these dominated tuples are removed and t is inserted into memory buffer; and 3) t is incomparable with all tuples in memory buffer. If there is enough room in memory buffer, p is inserted into memory buffer. Otherwise, p is written to a temporary file in disk. At the end of scanning, the candidate skyline tuples in memory buffer inserted before the temporary file is generated are part of skyline, and the remaining candidate skyline tuples have to compare with the tuples in temporary file. Here, the temporary file is the input table and is processed as above.

Chomicki et al. [12] devise a skyline algorithm SFS. Like BNL, SFS is also a multipass algorithm and maintains candidate skyline tuples in memory buffer. However, SFS first sorts the table in certain order compatible with the skyline criteria (*sorting phase*). Then, during scanning the table, SFS performs a similar dominance checking as BNL (*skyline-filter phase*). It is guaranteed for SFS that any candidate tuple is a part of skyline if it is inserted into memory buffer. Therefore, SFS has much less bookkeeping overhead than BNL and is insensitive to how the table is ordered originally.

Godfrey et al. [19] propose a more efficient skyline algorithm LESS to improve SFS. Similar to SFS, LESS first sorts the table in certain order compatible with the skyline criteria. However, LESS makes two major changes in disk-based sorting operation: 1) during the partitioning stage, LESS uses an *elimination-filter (EF)* buffer to discard the dominated tuples directly; 2) during the merging stage, LESS invokes the first skyline-filter phase as in SFS. LESS integrates sorting and skyline processing. It has all of SFS's benefits without additional disadvantages and consistently outperforms SFS. LESS also has BNL's advantages, but effectively none of its disadvantages. LESS does not need the bookkeeping overhead, and it requires much less cost for sorting than SFS because many tuples are discarded by EF buffer. Furthermore, LESS is invulnerable to how the table is ordered originally.

Bartolini et al. [1] develop SalSa algorithm based on SFS to exploit the sorting of a table to order tuples so that only a subset of table needs to be examined for computing skyline results. SalSa first sorts the table in certain order as in SFS. It denotes by $\mathcal{U}$ all *unread tuples* in table. Initially, $\mathcal{U}$ represents all tuples in the table. Each time a new tuple p is read from $\mathcal{U}$, p is compared against the current skyline tuples in memory buffer as in BNL. SalSa makes use of a stop point p_{stop} to check whether it can terminate reading tuples. When the current tuple retrieved from $\mathcal{U}$ is inserted into memory buffer, this might trigger the update of p_{stop}. It is guaranteed that SalSa terminates if all tuples in $\mathcal{U}$ are dominated by p_{stop}, and memory buffer keeps the skyline results.

To sum up, the current generic skyline algorithms have to scan the entire table at least once to return the skyline results. On big data, the algorithms will incur high I/O cost.

3.3 Summary

Of course, there are many other skyline algorithms in different applications, such as personalized skyline [2], [36], metric skyline [10], distributed skyline [11], [13], [21], [37]. In this paper, we focus on skyline query on a standalone computer. The algorithm proposed in this paper combines the advantages of index-based algorithms and generic algorithms, and overcomes their disadvantages. It preconstructs data-structures with low space overhead. By the data-structures, the algorithm only involves a small part of table to return the skyline results.

4 The SSPL Algorithm

This section first introduces the data-structures required by SSPL (see Section 4.1), then describes the overview of the SSPL algorithm (see Section 4.2), next shows how to perform pruning (see Section 4.3), followed that presents the implementation and analysis of SSPL (see Sections 4.4 and 4.5), and finally introduces how to extend SSPL to cover other cases (see Section 4.6).

4.1 Sorted Positional Index List

Definition 4.1 (Positional Index). *Given a table T, the positional index (PI) of $t \in T$ is i if t is the ith tuple in T.*

We denote by $T(i)$ the tuple in T with its $PI = i$, and by $T(i)[j]$ the jth attribute of $T(i)$. The execution of SSPL requires *sorted positional index lists*. Given a table $T(A_1, A_2, \ldots, A_M)$, we maintain a sorted positional index list L_j for each attribute A_j $(1 \leq j \leq M)$. L_j keeps the positional index information in T and is arranged in ascending order of A_j, that is $\forall i_1, i_2 (1 \leq i_1 < i_2 \leq n)$, $T(L_j(i_1))[j] \leq T(L_j(i_2))[j]$.

The sorted positional index lists are constructed as follows: First, table T is kept as a set of column files $CS = \{C_1, C_2, \ldots, C_M\}$ [29]. The schema of each column file C_j is $C_j(PI, A_j)$ $(1 \leq j \leq M)$, here PI represents the positional index of the tuple in T and A_j is the corresponding attribute value of $T(PI)$. Then, each column file C_j is sorted in ascending order according to A_j. Because SSPL only involves PI field of column files, the PI values in column files are retained and kept as *sorted positional index lists*. Here we compare the sorted positional index lists with the indexes used in tree-based algorithms briefly. SSPL constructs a sorted positional index list for each attribute, only M lists are needed. SSPL reduces the space overhead of data-structures from exponential to linear. More importantly, the processing of SSPL can cover all attributes, rather than limited to a small and selective set of attribute combinations in tree-based algorithms.

It is noted that read/append-only is an important characteristic of big data, and update is performed in periodic and batch mode. Therefore, sorted positional index lists are worth pre-computing and will be used repeatedly until the next update. And when update operation begins, sorted positional index lists can be updated by merging the corresponding column files in big old data and relatively much smaller new data.

4.2 The Overview of Algorithm

According to Section 2, it is known that the proportion of skyline results among all tuples is rather small. It means that

TABLE 1
Summary of Symbols

Symbol	Meaning
T	the table for skyline query
L_j	the sorted positional index list for attribute A_j
$AS_{skyline}$	skyline criteria
n	the tuple number in T
m	the size of $AS_{skyline}$
d	scan depth in phase 1
HT	hash table for candidate PIs in phase 1
SET_{cand}	candidate positional index set
NUM_{cand}	the size of SET_{cand} without pruning
R_k	the candidate positional indexes lying in Region k
REG_{cand}	the regions containing candidate positional indexes
$REG_{concern}$	the concerned regions in early pruning
V_{reg}	the volume of region reg
pf_{early}	the pruning ratio of early pruning
pi_{all}	the first positional index seen in $L_1, \ldots, L_m$ in phase 1

the overwhelming majority of retrieved tuples in generic algorithms are not the skyline results. Thus, the intuitive idea of SSPL is to omit the tuples that are not part of skyline results as much as possible. In this way, the CPU cost and I/O cost can be reduced significantly. The symbols used in this paper are shown in Table 1. SSPL consists of two phases:

- Phase 1: SSPL retrieves the sorted positional index lists corresponding to $AS_{skyline}$ to obtain candidate positional index set SET_{cand}.
- Phase 2: By SET_{cand}, SSPL performs a selective and sequential scan on the table to compute skyline results.

Next, we introduce two phases, respectively.

4.2.1 Phase 1: Obtaining Candidate Positional Index Set

Given $AS_{skyline} = \{A_1, A_2, \ldots, A_m\}$, SSPL needs to retrieve the involved sorted positional index list set $LS = \{L_1, L_2, \ldots, L_m\}$ in phase 1, whose process is shown as below:

- SSPL retrieves the lists in LS in a round-robin fashion. For each seen candidate positional index *pi*, SSPL checks whether hash table *HT* (initially empty) contains a record with key *pi*. If there is no such record, a new record ($key = pi, value = 1$) is inserted into *HT*. Here *key* represents the candidate positional index and *value* represents the occurrence number of *key* during retrieval in phase 1. Otherwise, if *HT* contains a record ($key = pi, value$), SSPL updates the record to be ($pi, value + 1$), i.e., the occurrence number of *pi* increases by 1.
- The retrieval operation continues until current candidate positional index *pi* occurs m times. That is, $\exists t \in T$ and $t = T(pi)$, *pi* has been retrieved in $L_1, L_2, \ldots, L_m$. Then, phase 1 ends and Theorem 4.1 proves that all tuples whose positional indexes are not contained in *HT* are not part of skyline.

Here it should be pointed out that, the candidate positional index *pi* retrieved in phase 1 is the positional index in T, rather than the positional index in sorted positional index list itself. For L_j, if its ith element is candidate positional index pi, i.e., $L_j(i) = pi$, it means that the value of $T(pi)[j]$ is in the ith position when the values of A_j are sorted in ascending order.

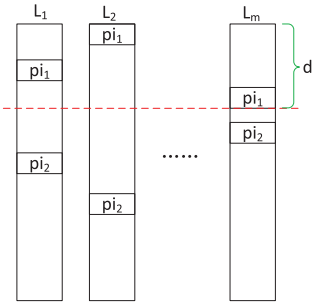


Fig. 1. The illustration for Theorem 4.1.

Theorem 4.1. *$\exists t_1 \in T$ and $t_1 = T(pi_1)$, if pi_1 is the first candidate positional index retrieved in $L_1, L_2, \ldots, L_m$ in phase 1, then it is guaranteed that skyline results are contained in the tuples whose positional indexes are kept in HT.*

Proof. $\forall t_2 \in T$ and $t_2 = T(pi_2)$, if pi_2 is not seen during the retrieval before pi_1 has been retrieved in $L_1, L_2, \ldots, L_m$, then pi_2 is not kept in *HT*. Because L_j is sorted in ascending order of A_j, as shown in Fig. 1, $\forall j (1 \leq j \leq m)$, $t_1[j] \leq t_2[j]$, $t_1 \succ t_2$.[1] That is, t_1 dominates any tuple in T whose positional index is not kept in *HT*. □

As illustrated in Fig. 1, we denote by d scan depth of $LS = \{L_1, L_2, \ldots, L_m\}$, that is the number of elements retrieved in each list at the end of phase 1. To facilitate analysis of scan depth, Assumption 4.1 is made in this part. The correlated and other distributions will be dealt with in Section 4.6.

Assumption 4.1. *The attributes in $AS_{skyline}$ are distributed uniformly and independently.*

Under Assumption 4.1, $\forall t \in T$ and $t = T(pi)$, $P(pi \in L_j(1 \ldots d)) = \frac{d}{n}$, the probability of *pi* lying in the first d elements of $L_1, L_2, \ldots, L_m$ is $p = (\frac{d}{n})^m$, here $L_j(1 \ldots d)$ represents the first d elements in L_j. We say that event S is successful if *pi* lies in the first d elements of $L_1, L_2, \ldots, L_m$, the number of successful events follows a binomial distribution $BD(n, p)$. According to *De Moivre-Laplace Theorem* [15], $BD(n, p)$ in this case can be replaced by a normal distribution $ND(\mu, \sigma^2)$, $\mu = np$, and $\sigma^2 = np(1-p)$. The validity of applying *De Moivre-Laplace Theorem* is proved in Theorem 4.2.

Theorem 4.2. *Under Assumption 4.1, the application of* De Moivre-Laplace Theorem *on $BD(n, p)$ is valid.*

Proof. It is known that *De Moivre-Laplace Theorem* is valid only when all samples from the unique binomial distribution are drawn independently. Under Assumption 4.1, $\forall t_1, t_2 \in T$, $t_1 = T(pi_1)$, and $t_2 = T(pi_2)$, the position of pi_1 in L_i does not affect the position of pi_2 in L_j $(i \neq j)$ for independent distribution. Besides, the position of pi_1 in L_i does not affect the position of pi_2 in L_i also. The statement can be proved that: given $t_2[i] \in C_i$ and the A_i attributes of $n-1$ tuples except for $t_2[i]$ in C_i have been arranged in ascending order, no matter where $t_1[i]$ is put in the sorted sequence, $t_2[i]$ can be located in any of n intervals generated by the A_i values of $n-1$ tuples with equal probability ($\frac{1}{n}$) for

1. In this paper, we consider that the tuples are in general position [28].

uniform distribution. In a word, the successful event for pi_1 is independent of that for pi_2. □

Theorem 4.3. *$d = n \times (\frac{-b+\sqrt{b^2-4a}}{2a})^{\frac{1}{m}}$ guarantees with a probability of 0.999968[2] that the number of the successful S is no less than 1 in which $a = n^2 + 16n$ and $b = -18n$.*

Proof. It is known that $\int_{\mu-4\sigma}^{+\infty} ND(\mu, \sigma^2)dx = 0.999968$. If $1 = \mu - 4\sigma$, it is guaranteed with a probability of 0.999968 that x-coordinate value of $ND(\mu, \sigma^2)$ is no less than 1, i.e., the number of the successful S is no less than 1. $(1 = \mu - 4\sigma) \Rightarrow (n^2 + 16n)p^2 - 18np + 1 = 0$. By solving this equation, we obtain $p = \frac{-b+\sqrt{b^2-4a}}{2a}$ in which $a = n^2 + 16n$ and $b = -18n$. (We reject another solution of this equation which is wrong obviously.) By $p = (\frac{d}{n})^m$, we get $d = n \times [\frac{-b+\sqrt{b^2-4a}}{2a}]^{\frac{1}{m}}$. □

By use of scan depth d obtained in Theorem 4.3, we compute the number NUM_{cand} of the candidate positional indexes maintained in *HT* at the end of phase 1. $\forall t \in T$ and $t = T(pi)$, $P(pi \in L_j(1 \ldots d)) = \frac{d}{n} (1 \leq j \leq m)$. Let $NUM(t, d)$ be the number of t's positional index pi satisfying $pi \in L_j(1 \ldots d)$, $NUM(t, d)$ follows a binomial distribution $BD_2(m, \frac{d}{n})$. We have: $P(NUM(t, d) = k) = \binom{m}{k} \times (\frac{d}{n})^k \times (1 - \frac{d}{n})^{m-k}$. If we do not perform any pruning operation, the number of the candidate positional indexes maintained in *HT* at the end of phase 1 is

$$NUM_{cand} = \sum_{k=1}^{m} [n \times P(NUM(t, d) = k)].$$

As shown in Theorem 4.1, SSPL returns skyline results by retrieving NUM_{cand} tuples in T. Let r_{cand} be the ratio of the candidate tuple number to all tuple number:

$$r_{cand} = \frac{NUM_{cand}}{n} = \sum_{k=1}^{m} P(NUM(t, d) = k) = 1 - \left(1 - \frac{d}{n}\right)^m.$$

The size of skyline criteria is typically small, and the relative value of scan depth d to tuple number n is small, so r_{cand} is small also. For example, given $n = 10^9, m = 4$, $r_{cand} = 0.0078$, SSPL just needs to involve 0.78 percent of tuples in T. Compared to at least once scan on the entire table in generic algorithms, SSPL reduces I/O cost significantly (the I/O cost involved in phase 1 is trivial compared to that in phase 2).

At the end of phase 1, SSPL keeps the candidate positional indexes in *HT* in set SET_{cand}, and sorts SET_{cand} in ascending order. The positional indexes in SET_{cand} correspond to the tuples retrieved in phase 2.

4.2.2 Phase 2: Retrieving the Skyline Results

In phase 2, SSPL retrieves the tuples in T whose positional indexes are contained in SET_{cand}. A sequential and selective scan is required to obtain the specified tuples in T since the elements in SET_{cand} are arranged in ascending order. Let T_{sub} be the subset of tuples in T specified by SET_{cand}. Phase 2 can be treated as a normal skyline processing on T_{sub} whose I/O cost is much lower due to much fewer tuples are involved. SSPL adopts current external skyline algorithms in phase 2. In this paper, we choose LESS to process skyline on T_{sub}.

2. We will see that the practical probability is even much higher.

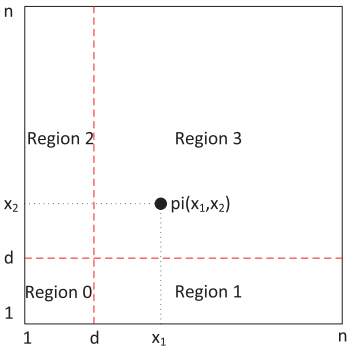


Fig. 2. The partitioning of coordinate space.

4.3 Pruning

Although the ratio r_{cand} of the candidate tuples to all tuple is small, the absolute number of the candidate tuples retrieved in phase 2 is still large. By observing the formula of NUM_{cand}, it is found that the number of tuples retrieved by SSPL increases exponentially as m increases, and increases polynomially as n increases. In this section, we consider how to improve SSPL further by pruning the candidate positional indexes in phase 1. Pruning operation consists of early pruning and late pruning, which will be presented below.

4.3.1 Early Pruning (EP)

EP performs pruning on each candidate positional index seen in phase 1. SSPL splits coordinate space into 2^m regions by the computed scan depth. Fig. 2 shows an instance of splitting coordinate space with skyline criteria $AS_{skyline} = \{A_1, A_2\}$.

$\forall t \in T$ and $t = T(pi)$, each candidate positional index pi corresponds to a point in m-dimensional coordinate space, whose jth dimension is the positional index of pi in L_j itself. That is, if $L_j(x_j) = pi (1 \leq j \leq m)$, the coordinate of pi in coordinate space is $(x_1, x_2, \ldots, x_m)$. As shown in Fig. 2, $L_1(x_1) = pi$ and $L_2(x_2) = pi$, the coordinate of pi is (x_1, x_2).

The scan depth d splits each dimension into two intervals $[1, d]$ and $[d, n]$ (Since $|T| = n$, we do not consider coordinates larger than n). *Region* k $(0 \leq k \leq 2^m - 1)$ can be determined by the value of k, let $k = (b_m b_{m-1} \ldots b_1)_2$ ($b_j = 0$ *or* 1). The projection of *Region* k on the jth $(1 \leq j \leq m)$ dimension is interval $[1, d]$ ($b_j = 0$) or $[d, n]$ ($b_j = 1$).

Let $R_k (0 \leq k \leq 2^m - 1)$ be the candidate positional indexes whose coordinates lie in *Region* k, we have $\sum_{k=0}^{2^m-1} |R_k| = n$. Note that *Region* 0 represents the bottom-left region that contains coordinate origin, while *Region* $(2^m - 1)$ represents the top-right region opposite to *Region* 0, which is illustrated in Fig. 2. According to the definition of scan depth, R_0 contains at least one candidate positional index. Therefore, the tuples specified by R_{2^m-1} are not part of skyline, and SSPL maintains the candidate positional index set $SET_{cand} = \bigcup_{k=0}^{2^m-2} R_k$.

Although SET_{cand} contains candidate positional indexes from $2^m - 1$ regions, *EP* only pays attention to m regions $REG_{concern} = \{Region\ [(2^m - 1) \oplus (1 \ll (j - 1))]\ (1 \leq j \leq m)\}$, each of which has the characteristics: *projections of the region on all dimensions are interval* $[d, n]$ *but one is* $[1, d]$. Here $\oplus$ is the bitwise XOR operation. These m regions contain the overwhelming majority of the candidate positional indexes in

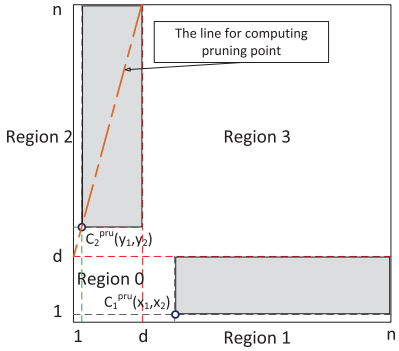


Fig. 3. An instance of early pruning.

SET_{cand} and to consider them only will efficiently simplify the processing of *EP*. Let $REG_{cand} = \{Region\ k\ (0 \le k \le 2^m - 2)\}$ represent the regions containing all candidate positional indexes, and let $f_{concern}$ be the ratio of the candidate positional indexes in $REG_{concern}$ to those in REG_{cand}. We compute $f_{concern}$ by volume ratio of $REG_{concern}$ to REG_{cand}. Also Assumption 4.1 is made in this part. Let V_{reg} be the volume of region *reg*,

$$f_{concern} = \frac{V_{REG_{concern}}}{V_{REG_{cand}}} = \frac{m \times \frac{d}{n} \times \left(1 - \frac{d}{n}\right)^{m-1}}{1 - \left(1 - \frac{d}{n}\right)^m}.$$

Here, we adopt *the binomial theorem* to clarify the formula of $f_{concern}$ to get an intuitive impression for its value. According to the theorem, $(x+y)^m = \sum_{k=0}^{m} \binom{m}{k} \times x^k \times y'^{m-k}$. We substitute $x = \frac{d}{n}$ and $y = 1 - \frac{d}{n}$ into the theorem and we have: $1 - (1 - \frac{d}{n})^m = m \times (\frac{d}{n}) \times (1 - \frac{d}{n})^{m-1} + \sum_{k=2}^{m} \binom{m}{k} \times (\frac{d}{n})^k \times (1 - \frac{d}{n})^{m-k}$.

By Theorem 4.3, $\frac{d}{n} \approx (\frac{c}{n})^{\frac{1}{m}}$, $9 \le c \le 18$. On big data, $\frac{d}{n}$ is close to 0, i.e., $\sum_{k=2}^{m} \binom{m}{k} \times (\frac{d}{n})^k \times (1 - \frac{d}{n})^{m-k}$ is rather small. Obviously, $f_{concern}$ is close to 1.

$$\forall region\ [(2^m - 1) \oplus (1 \ll (j-1))] \in REG_{concern} (1 \le j \le m),$$

its bottom-left coordinate $C_j^{bl} = (d, \ldots, d, 1, d, \ldots, d)$ and top-right coordinate $C_j^{tr} = (n, \ldots, n, d, n, \ldots, n)$. Here, the values in coordinate of $C_j^{bl}(C_j^{tr})$ are same except for the jth dimension. We denote by $region(C_j^{bl}, C_j^{tr})$ the region with bottom-left coordinate C_j^{bl} and top-right coordinate C_j^{tr}. As shown in Fig. 3, *EP* finds a point $C_j^{pru} = (y_1, y_2, \ldots, y_m)$ in $Region\ [(2^m - 1) \oplus (1 \ll (j-1))]$ which satisfies the conditions:

- $Region(C_j^{bl}, C_j^{pru})$ contains at least a candidate positional index.
- Maximize $r_j^{pru} = V_{region(C_j^{pru}, C_j^{tr})} / V_{region(C_j^{bl}, C_j^{pru})}$.

This can be explained as follows: The tuples specified by $region(C_j^{pru}, C_j^{tr})$ obviously are dominated by those specified by $region(C_j^{bl}, C_j^{pru})$, and to maximize r_j^{pru} is to maximize the number of the candidate positional indexes pruned in $region\ [(2^m - 1) \oplus (1 \ll (j-1))]$.

Here an efficient method is proposed to compute C_j^{pru}. We first analyze the relationship among the coordinate values of C_j^{pru}, and begin with a simplified problem which determines $C_{cell}^{pru} = (a_1, a_2, \ldots, a_m)(0 \le a_j \le 1)$ in m-dimensional cell region $Region_{cell}(00\ldots0, 11\ldots1)$ satisfying the conditions:

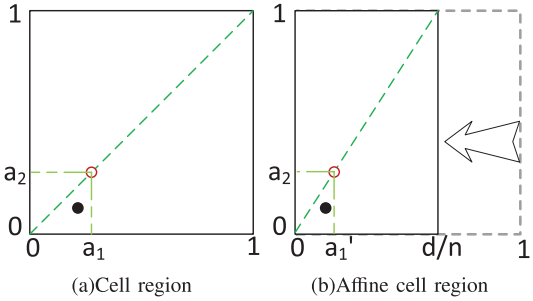


Fig. 4. The position of pruning point in cell region.

$$\begin{cases} \prod_{k=1}^{m} a_k = \lambda\ (\lambda\ is\ a\ constant\ and\ 0 < \lambda < 1), \\ maximize \prod_{k=1}^{m} (1 - a_k). \end{cases}$$

According to inequality of arithmetic and geometric means, for any list of k nonnegative real numbers $x_1, x_2, \ldots, x_k$, $\sqrt[k]{x_1 \times x_2 \times \cdots \times x_k} \le \frac{x_1 + x_2 + \cdots + x_k}{k}$ [34]. We have

$$\prod_{k=1}^{m} (1 - a_k) \le \left(\frac{m - \sum_{k=1}^{m} a_k}{m}\right)^m \le \left(\frac{m - m \times \left(\prod_{k=1}^{m} a_k\right)^{\frac{1}{m}}}{m}\right)^m.$$

When $a_{k_1} = a_{k_2} (1 \le k_1 \ne k_2 \le m)$, the equality holds. The satisfied C_{cell}^{pru} is on the diagonal $(00\ldots0, 11\ldots1)$ of $Region_{cell}$ as shown in Fig. 4a.

Next, we conduct an affine transformation [5] on $Region_{cell}$ to get $Region_{afcell}$. As shown in Fig. 4b, $\forall (e_1, e_2, \ldots, e_m) \in Region_{cell}$, it is transformed to $(e'_1, e'_2, \ldots, e'_m) \in Region_{afcell}$ by affine transformation, here $e'_j = affunc(e_j)\ (1 \le j \le m)$ and $affunc$ is the used affine function. It is known that *affine property remains invariant in affine transformation* [5]. Therefore, the satisfied point in $Region_{afcell}$ corresponding to C_{cell}^{pru} is also on the diagonal of $Region_{afcell}$. By affine transformation of expansion and translation of coordinate, the affine property still holds in $Region\ [(2^m - 1) \oplus (1 \ll (j-1))]$. That is, the satisfied pruning point $C_j^{pru} = (y_1, y_2, \ldots, y_m)$ is on the diagonal of $Region\ [(2^m - 1) \oplus (1 \ll (j-1))]$.

According to Theorem 4.3, the satisfied pruning point in cell region $Region_{cell}$ is: $C_{cell}^{pru} = \{\frac{d'}{n'}, \ldots, \frac{d'}{n'}\}$, here n' is the tuple number in $Region_{cell}$ and d' is scan depth in $Region_{cell}$ computed by Theorem 4.3. n' can be computed by the volume proportion of $Region\ [(2^m - 1) \oplus (1 \ll (j-1))]$ to coordinate space. $\forall (e_1, e_2, \ldots, e_m) \in Region_{cell}$, we conduct an affine transformation on this point to get $(e''_1, e''_2, \ldots, e''_m) \in Region\ [(2^m - 1) \oplus (1 \ll (j-1))]$, in which

$$e''_k = \begin{cases} (n - d) \times e_k + d\ (k \ne j), \\ d \times e_k + 1\ (k = j). \end{cases}$$

Thus, the pruning point in $region\ [(2^m - 1) \oplus (1 \ll (j-1))]$ is

$$C_j^{pru}[k] = \begin{cases} (n - d) \times \frac{d'}{n'} + d\ (k \ne j), \\ d \times \frac{d'}{n'} + 1\ (k = j). \end{cases}$$

EP computes the pruning point for each region in $REG_{concern}$.

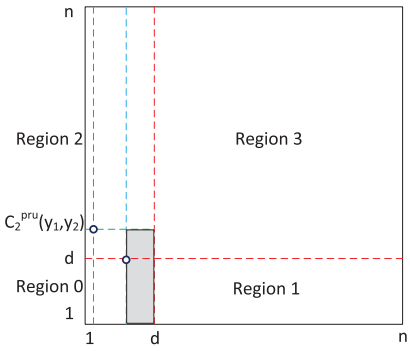


Fig. 5. The pruned region of late pruning.

$\forall pi = L_j(i)$ $(1 \le j \le m, 1 \le i \le n)$, SSPL performs *EP* operation on each candidate positional index *pi* seen in phase 1. The pruning rule for *EP* is listed as follows:

The Rule for Early Pruning:
1. If $i \le C_j^{pru}[j]$, the candidate positional index *pi* cannot be pruned.
2. Otherwise, for $i > C_j^{pru}[j]$, if $\exists k(1 \le k \le m \text{ and } k \ne j)$ $pi = L_k(i_k)$ and $i_k \le C_j^{pru}[k]$, *pi* cannot be pruned.
3. In other cases, *pi* can be pruned.

The intuitive idea of *EP* is that $\forall pi = L_j(i)$ retrieved in phase 1, if the coordinate of *pi* lies in $Region(C_j^{pru}, C_j^{tr})$ in $Region\ [(2^m - 1) \oplus (1 \ll (j-1))]$, then $T(pi)$ is not part of skyline. As shown in Fig. 3, the retrieved candidate positional indexes whose corresponding coordinates lie in the shadow regions can be discarded directly. Next, we analyze the theoretical effect of *EP*. Let $pf_{REG_{concern}}$ be the pruning ratio of the candidate positional indexes in $REG_{concern}$. $pf_{REG_{concern}}$ can be computed by the volume ratio of the regions. Let V_{reg} be the volume of region *reg*. According to the rule for early pruning,

$$pf_{REG_{concern}} = V_{\bigcup_{j=1}^{m} Region(C_j^{pru}, C_j^{tr})} / V_{REG_{concern}} = \left(1 - \frac{d'}{n'}\right)^m.$$

Considering $Region\ [(2^m - 1) \oplus (1 \ll (j-1))]$ $(1 \le j \le m)$, the pruning ratio is $(1 - \frac{d'}{n'})^m$ which can be deduced from the original coordinate space. Thus, we obtain the formula of $pf_{REG_{concern}}$. By use of $f_{concern}$ and $pf_{REG_{concern}}$, we get the pruning ratio pf_{early} of *EP*:

$$\begin{aligned} pf_{early} &= f_{concern} \times pf_{REG_{concern}} \\ &= \frac{m \times \frac{d}{n} \times \left(1 - \frac{d}{n}\right)^{m-1} \times \left(1 - \frac{d'}{n'}\right)^m}{1 - \left(1 - \frac{d}{n}\right)^m}. \end{aligned}$$

Considering the size m of skyline criteria is typically small, and the tuple number $n(n')$ on big data is large, based on the similar analysis as $f_{concern}$, *EP* can prune most of the candidate positional indexes in phase 1.

4.3.2 Late Pruning (LP)

This part introduces late pruning which is executed on the candidate positional indexes at the end of phase 1.

Theorem 4.3 proves that when d elements are retrieved from $L_1, L_2, \ldots, L_m$, respectively, there occurs at least one candidate positional index *pi* which is seen in the first d elements in $L_1, L_2, \ldots, L_m$. Usually, *pi* is located in $L_1, L_2, \ldots, L_m$ at different positions. Because SSPL retrieves the involved lists in a round-robin fashion in phase 1, there are still a part of records in *HT* which can be pruned further and cannot be detected by *EP*. To prune these candidates is the task of *LP*.

As shown in Fig. 5, the shadow region is one of areas for late pruning. Let pi_{all} be the first positional index which is seen in $L_1, L_2, \ldots, L_m$ during the retrieval in phase 1. The rule for late pruning is listed as follows.

The Rule for Late Pruning:
At the end of phase 1 in SSPL, the candidate positional index pi_{cand} can be pruned if it satisfies condition: all positional indexes of pi_{cand} in $L_1, L_2, \ldots, L_m$ themselves seen during the retrieval are no less than those of pi_{all}.

According to the similar proof in Theorem 4.1, the candidate positional indexes pruned by late pruning correspond to the tuples which are dominated by the tuple specified by pi_{all}. The performance of *LP* depends on the actual positions of pi_{all} in $L_1, L_2, \ldots, L_m$. Thus, we do not provide the theoretical analysis of the pruning effect of *LP* here.

4.4 Algorithm Implementation

Although it is nontrivial to implement early pruning practically, SSPL achieves the goal by employing EGBFT for a quick response to membership checking on *PI* set in exponential-gap positional index range of the involved lists.

Definition 4.2 (Exponential-Gap Bloom Filter Table). *Given sorted positional index list L_j with n elements, $EGBFT_j$ is exponential-gap bloom filter table for L_j, if $EGBFT_j$ satisfies: 1) $|EGBFT_j| = log_2 n$, 2) $EGBFT_j(i)$ is a bloom filter constructed on PI attribute from $L_j(1)$ to $L_j(2^i)(0 \le i \le log_2 n)$.*

The disk space required by $EGBFT_j$ depends on tuple number n and false positive rate *fpr* of bloom filter. $EGBFT_j(i)$ is a bloom filter on 2^i elements with a-bit bit-vector and b independent hash functions. If $b = \frac{a \ln 2}{2^i}$, the minimum $fpr = (\frac{1}{2})^b$, the optimal length of bit-vector is $a = \frac{2^i \times log_2(\frac{1}{fpr})}{ln2}$ [6]. Let $SIZE_{EGBFT_j}$ be size of $EGBFT_j$. Given $|EGBFT_j| = log_2 n$, $SIZE_{EGBFT_j} = \sum_{i=0}^{log_2 n} \frac{2^i \times log_2(\frac{1}{fpr})}{8 \times ln2} = \frac{(2n-1) log_2(\frac{1}{fpr})}{8 \times ln2}$ has a linear relationship with n if *fpr* is fixed. We set $fpr = 0.001$ in this paper, $SIZE_{EGBFT_j}$ is less than 45 percent of the size of L_j. SSPL preconstructs EGBFT on each sorted positional index list.

SSPL first obtains scan depth d according to Theorem 4.3. Then it computes pruning point set $PS_{pru} = \{C_k^{pru}(1 \le k \le m)\}$ for m regions in $REG_{concern}$. For the jth dimension $(1 \le j \le m)$, we get its pruning coordinate $PC_j = max\{C_k^{pru}[j]\ (1 \le k \le m)\}$ and remaining coordinate $RC_j = min\{C_k^{pru}[j]\ (1 \le k \le m)\}$. *Note that the coordinate here is the positional index in sorted positional index list itself as depicted in Section 4.3.1.* SSPL loads $EGBFT_j(i)$ for each L_j, here $i = \lceil log_2 PC_j \rceil$.[3] The binary logarithm

3. This is the reason why the practical probability is much higher than the theoretical value in Theorem 4.3.

function used by i corresponds to the positional index of the required EGBFT tuple. It is a good space/efficiency tradeoff to construct EGBFT for early pruning. Of course, i can be computed by another function of PC_j if the data structure for early pruning is constructed by a different method. During the retrieval operation in phase 1, SSPL utilizes $EGBFT_j(i)(1 \le j \le m)$ to perform early pruning on each candidate positional index $pi = L_j(pil)$ seen in phase 1, *pil* is the positional index of *pi* in L_j itself. The pseudocode of early pruning is shown in Algorithm 1. Algorithm 1 first checks whether *pil* is less than RC_j. If so, *pi* cannot be pruned according to *item* 1 in the rule for early pruning. Otherwise, Algorithm 1 checks whether *pi* is contained in the dominated region according to *item* 2 and *item* 3 in the rule for early pruning. If so, *pi* can be discarded. If not, *pi* is a candidate positional index and kept in *HT*.

Algorithm 1. EarlyPruning(*j*, pil, pit).

// testInBF(bf, pi) checks whether pi belongs to S on which
// bloom filter bf is constructed, true is in, false is not.
// j is the index for sorted-positional-index-list
// pil is the positional index in sorted-positional-index-list
// pit is the candidate positional index for T
// return: true - pit can be pruned, false - pit cannot be pruned

1: int $index_j = \lceil log_2 PC_j \rceil$//($PC_j = max\{C_k^{pru}[j]\ (1 \le k \le m)\}$)
2: **if** $pil \le RC_j$ **then**
3: return false //($RC_j = min\{C_k^{pru}[j]\ (1 \le k \le m)\}$)
4: **end if**
5: **for** $k = 1$ to m **do**
6: **if** j == k **then**
7: continue
8: **end if**
9: boolean inflag = testInBF($EGBFT_k(index_j)$, pit)
10: **if** (inflag) **then**
11: return false
12: **end if**
13: **end for**
14: return true

In phase 1, for each candidate positional index *pi* retrieved from L_j, SSPL maintains the positional index of *pi* in L_j itself in *HT*. At the end of phase 1, SSPL invokes late pruning whose pseudo-code is shown in Algorithm 2. Late pruning discards all candidate positional indexes whose corresponding tuples are dominated by the tuple specified by pi_{all}.

Algorithm 2. LatePruning().

// occurinfo is the positional index information of candidate PI in
// lists itself maintained during retrieval in phase 1

1: Iterating the elements in hash table *HT*
2: **while** HT.hasNext() **do**
3: Record rec = HT.next()
4: long pi = rec.PI
5: long[] pos = rec.occurinfo
6: boolean flag = true
7: **for** $j = 1$ to m **do**
8: **if** pos[j] $\le pi_{all}[j]$ **then**
9: flag = false
10: break
11: **end if**
12: **end for**
13: **if** flag **then**
14: rec is removed from *HT*
15: **end if**
16: **end while**

By early pruning and late pruning, SSPL is implemented as follows: SSPL retrieves $L_1, L_2, \ldots, L_m$ in a round-robin fashion, and *EP* is executed on each candidate positional index seen in phase 1. The candidates which cannot be pruned are maintained in *HT*. The retrieval process continues until pi_{all} occurs. If the size of candidate positional indexes exceeds the maximum limit in memory buffer, the processing in LARA [25] is adopted to merge hash table by use of disk as exchange space. At the end of phase 1, SSPL invokes late pruning on *HT*. Next, SSPL enters phase 2 and retrieves the tuples with the specified positional indexes obtained in phase 1. Here, the existing external skyline algorithm is invoked to return skyline results. The pseudocode of SSPL is shown in Algorithm 3.

Algorithm 3. SSPL($T, L_1, L_2, \ldots, L_m$).

// T is the table on which skyline is performed
// L_j $(1 \le j \le m)$ is the sorted positional index list for A_j

1: long list-index = 0, boolean bStage1 = true
2: loop from line 3 to line 22
3: **if** bStage1 **then**
4: read ($pi_1, pi_2, \ldots, pi_m$) from $L_1, L_2, \ldots, L_m$
5: list-index + = 1
6: **for** $j = 1$ to m **do**
7: **if** EarlyPruning(j, list-index, pi_j) **then**
8: continue
9: **else**
10: maintain pi_j in *HT* with occurrence info
11: **if** pi_j.count == m **then**
12: bStage1 = false
13: LatePruning()
14: sort records in *HT* to SET in ascending order
15: break
16: **end if**
17: **end if**
18: **end for**
19: **else**
20: retrieve tuples in T with the positional indexes in SET
21: perform skyline on the retrieved tuples with the existing external algorithm, and return results
22: **end if**

Through the description of algorithm implementation, we know that the actual effect of early pruning will be a little worse than the theoretical effect in Section 4.3. The reason is that the pruning coordinate values of each pruning point are expanded to their minimum upper bounds of 2 to the power by use of exponential-gap bloom filters. However, the actual pruning effect of early pruning is still good enough, which will be verified by the experimental results in the next section.

4.5 Analysis

In this part, we discuss the performance guarantee for SSPL. We utilize the number of the retrieved tuples in phase 2 as a measure of performance, since it directly determines the I/O cost and CPU cost. For generic algorithm such as LESS, its expected computation time is *O(n)* [19]. By Theorem 4.3, $\frac{d}{n} = O(c^{\frac{1}{m}} \times n^{-\frac{1}{m}}), 9 \leq c \leq 18$. As shown in Section 4.2.1, the number of the candidate tuples in phase 2 without pruning is: $NUM_{cand} = n \times [1 - (1 - \frac{d}{n})^m] = n \times [m \times (\frac{d}{n}) \times (1 - \frac{d}{n})^{m-1} + \sum_{k=2}^{m} (\frac{m}{k}) \times (\frac{d}{n})^k \times (1 - \frac{d}{n})^{m-k}] = O(c^{\frac{1}{m}} m n^{1-\frac{1}{m}})(9 \leq c \leq 18)$. As shown in Section 4.3.1, the pruning ratio pf_{early} of early pruning is $\frac{m \times \frac{d}{n} \times (1-\frac{d}{n})^{m-1} \times (1-\frac{d'}{n'})^m}{1-(1-\frac{d}{n})^m}$. Here, $n' = n \times \frac{d \times (n-d)^{m-1}}{n^m}$. Similarly, $1 - pf_{early} = O(c^{\frac{1}{m}-\frac{1}{m^2}} m n^{\frac{1}{m^2}-\frac{1}{m}})$. The number of the tuples retrieved in phase 2 is not more than $NUM_{cand} \times (1 - pf_{early})$ (the late pruning is not considered here because its pruning ratio depends on the actual execution). Therefore, the expected computation time for SSPL is $O(c^{\frac{2}{m}-\frac{1}{m^2}} m^2 n^{1+\frac{1}{m^2}-\frac{2}{m}})$.

4.6 Extensions

If the involved attributes are correlated (positive correlation or negative correlation), early pruning cannot make use of the coordinates of the pruning points computed in Section 4.3.1 since Assumption 4.1 does not hold. However, late pruning still works here because it discards the candidate positional indexes whose corresponding tuples are dominated by the tuple specified by pi_{all} and is not affected by data distribution.

According to Section 4.4, it is found that the key for early pruning is to determine the remaining coordinate and pruning coordinate for each dimension. This paper proposes a new data structure remaining and pruning coordinate information table (*RPCIT*) to efficiently determine the required information for each dimension. It is noted that the size of skyline criteria is typically small in practical applications, thus we give an upper limit m_{max} for the size. Given the dimensionality M of a table T, the tuple number contained in RPCIT is $\sum_{m=2}^{m_{max}} (\frac{M}{m})$. Given any RPCIT tuple *rpci*, the positional index of *rpci* in RPCIT is represented by an M-bit bit-vector $b = (b_M b_{M-1} \dots b_1)_2$. If skyline criteria contains attribute $A_j (1 \leq j \leq M)$, $b_j = 1$, otherwise $b_j = 0$. For any tuple *rpci* with m one-bits in M-bit bit-vector, *rpci* contains $2m$ attributes $(RC_1, PC_1, RC_2, PC_2, \dots, RC_m, PC_m)$. Here, (RC_j, PC_j) are the remaining coordinate and pruning coordinate for the j^{th} dimension in skyline criteria.

Given skyline criteria $AS_{skyline}$ of size m, SSPL first gets the corresponding positional index *pi* in RPCIT

$$pi = (b_M b_{M-1} \dots b_1) b_i = \begin{cases} 0, & \text{if} A_i \notin AS_{skyline}, \\ 1, & \text{if} A_i \in AS_{skyline}. \end{cases}$$

After loading RPCIT(pi), we get RC_j and PC_j for the jth dimension in $AS_{skyline}$: $RC_j = RPCIT(pi).RC_j$ and $PC_j = RPCIT(pi).PC_j$. Then SSPL loads $EGBFT_j(\lceil log_2 PC_j \rceil)$ $(1 \leq j \leq m)$ for L_j. The other operation is similar as Algorithm 3.

The size of single RPCIT tuple whose M-bit positional index contains m one-bits is $16m$ bytes (its attributes are of type *long* here). Let $SIZE_{RPCIT}$ be size of $RPCIT$, then $SIZE_{RPCIT} = \sum_{m=2}^{m_{max}} [16m \times (\frac{M}{m})]$. As M increases, $SIZE_{RPCIT}$ increases quickly. However, the space requirement is acceptable when dimensionality is in moderate size. For example, when $M = 30$ and $m_{max} = 10$, $SIZE_{RPCIT} = 7.4G$. In the case of high dimensionality, RPCIT cannot maintain information about all attributes due to the limitation of the size. Here, we select a subset of attributes that are the most frequently involved in skyline criteria to construct RPCIT according to the size limitation. In this way, the constructed RPCIT not only meets the size limitation, but it also can process most of the queries. If some skyline queries cannot be covered by RPCIT, SSPL performs in the similar way as that in Assumption 4.1 except that *EP* is not invoked in phase 1.

For other data distributions, SSPL can be performed in the similar way as that in this subsection.

Although we focus on skyline query on a standalone computer in this paper, it is still interesting to discuss how SSPL can be adopted in distributed environment. As stated in [21], almost all existing distributed skyline algorithms contain three phases: local processing, query routing and result merging. Of course, SSPL can be used in local processing phase to compute the local skyline results, and be combined with *tuple filtering* and *peer pruning* to process distributed skyline query.

5 Performance Evaluation

5.1 Experimental Settings

To evaluate performance of SSPL, we implement it in Java with jdk-6u20-windows-x64. The experiments are executed on HP xw8600 (8×2.8 GHz Xeon CPU + 32G memory + 64 bit windows). The used data is stored in Seagate Barracuda LP ST32000542AS (2TB). We evaluate the performance of SSPL against LESS [19], ZSearch [24], and SUBSKY [31]. The reason for the choice is explained below. The skyline algorithms can be divided into two classes: index-based algorithms (such as ZSearch and SUBSKY) and generic algorithms (such as LESS). As we mentioned before, the practicality of index-based algorithms is seriously limited for their prohibitive precomputation cost and space overhead (exponential number of indexes have to be built for tree-based algorithms) and it is difficult (if not impossible) for index-based algorithms to build the necessary indexes to cover the queries posed on tables. Nevertheless, to verify completely the performance of SSPL, we still compare SSPL against ZSearch and SUBSKY, the former organizes and accesses tuples based on a Z-order space filling curve, the latter accesses the tuples in descending order of the specified distance function. We sort the tuples respectively for ZSearch and SUBSKY according to their requirement in the experiments. As to generic algorithms, LESS combines the advantages of SFS, BNL and M3 algorithm [4], and incurs less I/O cost than divide&conquer algorithms. In the experiments with settings in this section, we find that the performance of LESS is almost the same as BNL and M3, and is better than SFS and divide&conquer algorithms.

TABLE 2
Parameter Settings

parameter	the used values
Data volume (synthetic)	0.2TB - 1.0TB
Size of skyline criteria (synthetic)	2 - 5
Tuple length (synthetic)	160 bytes
PCC for correlated data (synthetic)	0.54
Data volume (real)	3.76GB - 18.8GB
Tuple length (real)	376 bytes

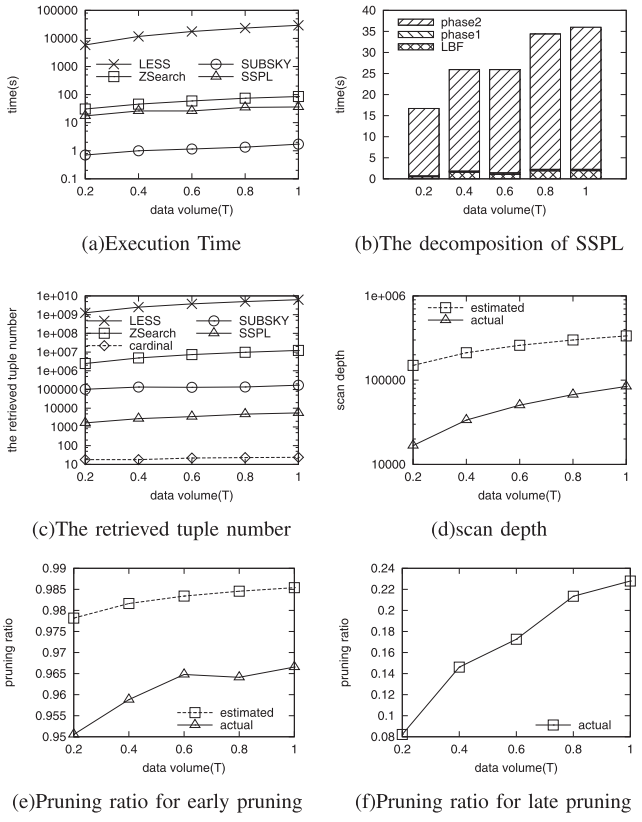
The experiments are executed on three data sets: two synthetic data sets(uniform distribution and correlated distribution) and a real data set. The used parameter settings are listed in Table 2. For uniform and correlated distributions, we synthesize tables with each tuple of 160 bytes. Each tuple has six value attributes (48 bytes) of type *long* and an overload of 112 bytes. For uniform distribution, the first five value attributes are generated uniformly and independently. For correlated distribution, the first two attributes are generated with Pearson Correlation Coefficient [16] (*PCC* for short) 0.54 (positive correlation). For synthetic data, the tuple numbers we consider in the experiments are 1.25B (billion), 2.5B, 3.75B, 5B, and 6.25B. That is, the data volumes considered are 0.2T, 0.4T, 0.6T, 0.8T, and 1.0T. The considered sizes of skyline criteria are 2, 3, 4, and 5. The real data set used is a part of communication log in certain month after anonymous processing from a communication corporation. Each tuple in communication log has 72 attributes and the tuple length is 376 bytes. In the experiments, we consider skyline query on real data set with varying data volumes and fixed size of skyline criteria.

The used sorted positional index lists are preconstructed at the beginning of experiments. Due to the space limitation, the experimental results are not illustrated in diagram. We briefly introduce the results here. It takes around 3,300 seconds to generate a sorted positional index list by sorting an attribute of 1.25B elements, and takes around 400 seconds to build the necessary EGBFT on the corresponding list. The required data-structures are obtained in the similar way.

The experimental results are computed by averaging three executions of each program and we restart computer between two consecutive executions to empty cache and memory.

5.2 Experiment 1: The Effect of Data Volume

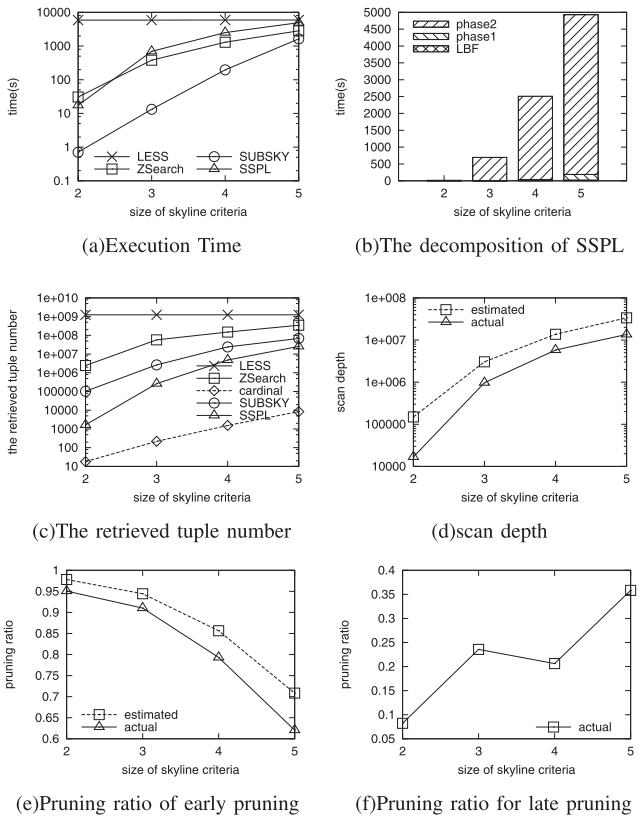
In experiment 1, we evaluate the performance of SSPL on varying data volumes and let the size of skyline criteria be 2. The attributes in skyline criteria here are generated uniformly and independently. The considered data volumes are 0.2TB, 0.4TB, 0.6TB, 0.8TB, and 1.0TB. As shown in Fig. 6a, SSPL runs 590.31 times faster than LESS on average. And the speedup ratio of SSPL to LESS becomes larger as the data volume is bigger. The speedup ratio is 337.06 when data volume is 0.2TB, while the ratio increases to 814.26 when data volume is 1TB. And as shown in Fig. 6a, SSPL runs faster than ZSearch but slower than SUBSKY. Fig. 6b shows the time decomposition of SSPL. The execution time of SSPL consists of three parts: loading EGBFT, phase 1 and phase 2. Here, phase 2 takes most of



(a)Execution Time (b)The decomposition of SSPL
(c)The retrieved tuple number (d)scan depth
(e)Pruning ratio for early pruning (f)Pruning ratio for late pruning

Fig. 6. The effect of data volume.

the time (over 91 percent). And next, loading EGBFT takes some time. The time for phase 1 is trivial compared to other two parts because it only retrieves a rather small proportion of the involved sorted positional index lists. As shown in Fig. 6b, the execution time of SSPL first increases when data volume rises from 0.2TB to 0.4TB, then it keeps basically unchanged when data volume rises to 0.6TB, followed that it increases considerably when data volume increases to 0.8TB, and finally it increases slightly when data volume rises to 1TB. This can be explained that, from 0.2TB to 0.4TB, the positional index of EGBFT to load increases by 1 to cover a larger range, and from 0.6TB to 0.8TB, the positional index of EGBFT to load increases by 1 again. As shown in Fig. 6c, SSPL retrieves 971395.29 times fewer tuples than LESS, 1895 times fewer tuples than ZSearch and 41 times fewer tuples than SUBSKY. The I/O cost in SSPL is reduced significantly, which also reflects the value of the precomputed data structures proposed in this paper. Here, the dotted line with diamond legend represents the cardinality of skyline query in experiment. Although SSPL retrieves fewer tuples than SUBSKY, the sequential seek operation for each retrieval makes the execution time of SSPL longer than SUBSKY, which only needs to read sequential tuples continuously. Fig. 6d shows the computed scan depth and the actual scan depth. It shows that our computation for scan depth in this paper is correct. Fig. 6e gives the theoretical pruning ratio and the actual pruning ratio of early pruning. In the figure, the dotted line represents the theoretical pruning ratio, while the solid line represents the actual pruning ratio. The actual pruning ratio of early pruning is a little worse than the estimated pruning ratio,



(a)Execution Time (b)The decomposition of SSPL

(c)The retrieved tuple number (d)scan depth

(e)Pruning ratio of early pruning (f)Pruning ratio for late pruning

Fig. 7. The effect of skyline criteria size.

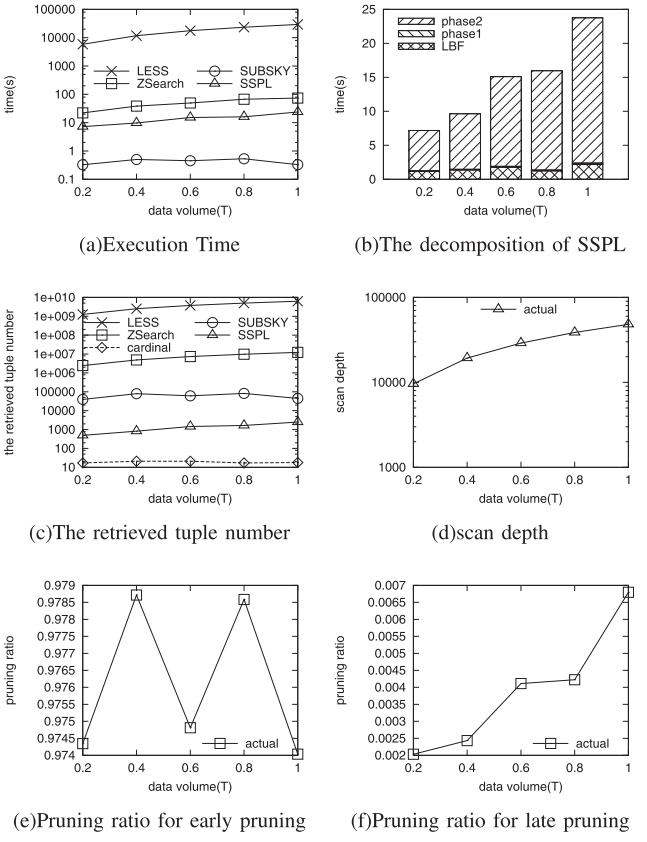
because the actual scan depth is smaller than the estimated one and the pruning coordinates of pruning points are expanded to their minimum upper bounds of 2 to the power. Early pruning ratio first rises when data volume changes from 0.2TB to 0.6TB. But the pruning ratio declines slightly when data volume changes from 0.6TB to 0.8TB since here the positional index of EGBFT to load increases by 1 which affects the effect of early pruning. Fig. 6f shows the effect of late pruning. And we can see that its pruning ratio increases as data volume becomes larger. For a larger data set, more candidate positional indexes are left after early pruning, this gives more room for late pruning to perform. The accuracy in the experiments is measured by the proportion of SSPL results appearing in LESS results (which is the accurate results of skyline query). For space limitations, we do not illustrate the accuracy here, but SSPL obtains its results correctly.

5.3 Experiment 2: The Effect of Skyline Criteria Size

In experiment 2, we fix data volume of 0.2TB and evaluate the performance of SSPL with varying sizes of skyline criteria (from 2 to 5). As shown in Fig. 7a, SSPL runs 87.26 times faster than LESS on average. The execution time of LESS is nearly unchanged since data volume is fixed and the I/O cost dominates its execution time. By contrast, the execution time of SSPL grows exponentially. The speedup ratio declines from 337.06 to 1.2. ZSearch and SUBSKY have the similar trend as SSPL. Overall, SSPL runs slower than ZSearch and SUBSKY, and when more attributes are involved in skyline criteria, their execution times grow fast and are close to LESS. As can be seen in Fig. 7b, the majority of execution time of SSPL is taken in phase 2. As more attributes are involved in skyline criteria, the time for phase 1 increases exponentially also and gradually overtakes the time for loading EGBFT. Fig. 7c illustrates the retrieved tuple number in phase 2. Obviously, it shows that the required number of the candidate tuples increases exponentially, but SSPL still retrieves 190093.48 times fewer tuples than LESS on average (48.62 times fewer even when size of skyline criteria is 5), 434 times fewer tuples than ZSearch and 20 times fewer tuples than SUBSKY. This can be explained by the formula of scan depth in Theorem 4.3 as shown in Fig. 7d. The actual scan depth in experiment 2 grows at an exponential rate. And this is also the reason for exponential growth of execution times in phase 1 and phase 2. Fig. 7e shows the estimated pruning ratio and actual pruning ratio of early pruning. As the formula for theoretical effect of early pruning in Section 4.3.1, the pruning ratio declines with more attributes involved in skyline criteria. The actual pruning ratio is worse than the theoretical one, but it still follows the trend analyzed in this paper. As indicated in Fig. 7f, the pruning ratio of late pruning increases with more attributes involved in skyline criteria except for the size being 3. This can be explained as below. The late pruning is used to discard the candidate positional indexes whose corresponding tuples are dominated by certain tuple t, here t is the tuple whose positional index is the first to be seen in all of the involved lists. Thus, it is a good indicator for the effect of late pruning to see the ratio $ratio_{le}$ of the volume of the region formed by the coordinate specified by t's positional index to the volume of the region formed by the actual scan depth. In experiment 2, $1\text{-}ratio_{le}$ first rises when skyline criteria size increases from 2 to 3, and then, $1\text{-}ratio_{le}$ declines gradually with larger size. This explains why pruning ratio of late pruning first rises, and then declines. As for another rise when size increases from 4 to 5, the worse early pruning effect makes more tuples left at the end of phase 1 and this gives more room for late pruning.

5.4 Experiment 3: The Effect of Correlated Attributes

In experiment 3, we fix the size of skyline criteria 2 and evaluate the performance of SSPL on correlated attributes with varying data volumes (from 0.2TB to 1.0TB). The attributes in skyline criteria are clustered along the diagonal line with *PCC* 0.54. As depicted in Fig. 8a, SSPL runs 1177.6 times faster than LESS. If the attributes in skyline criteria are positively correlated, we will find the candidate positional index which is seen in all of the involved lists earlier. The trend of execution time for SSPL, ZSearch, and SUBSKY is similar to that in experiment 1, but faster. Fig. 8b illustrates the decomposition of SSPL. RPCIT is preconstructed here for SSPL. The execution time of loading EGBFT is nearly the same as that in experiment 1, while the execution time for phase 1 and phase 2 is smaller. As shown in Fig. 8c, SSPL retrieves 2744375.7 times fewer tuples than LESS, 5353.8 times fewer tuples than ZSearch and 57.5 times fewer tuples than SUBSKY. Fig. 8d shows scan depth of SSPL, which increases linearly with larger data volume. The scan depth in experiment 3 is only half of that in experiment 1 because of the effect of positively correlated distribution. Fig. 8e

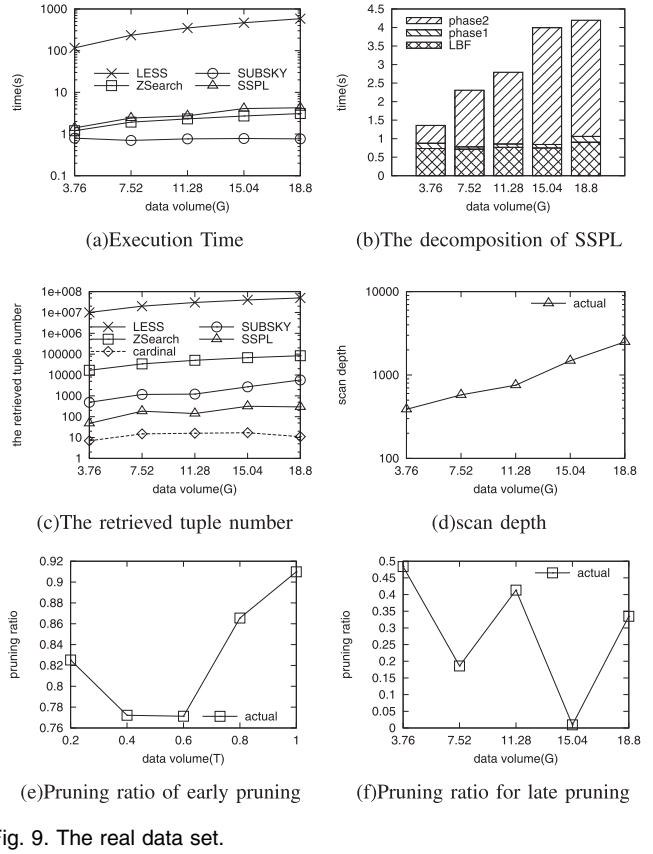


(a)Execution Time (b)The decomposition of SSPL

(c)The retrieved tuple number (d)scan depth

(e)Pruning ratio for early pruning (f)Pruning ratio for late pruning

Fig. 8. The effect of correlated attributes.



(a)Execution Time (b)The decomposition of SSPL

(c)The retrieved tuple number (d)scan depth

(e)Pruning ratio of early pruning (f)Pruning ratio for late pruning

Fig. 9. The real data set.

shows the pruning ratio of early pruning, whose values are larger than those in experiment 1. The reason is that the attributes with positive correlation tend to move towards the same direction, and this makes a better early pruning effect. The effect of late pruning is illustrated in Fig. 8f, it is shown that its pruning ratio is smaller than that in experiment 1 because of the effect of the positive correlation. But the pruning ratio of late pruning still increases linearly with the involved data volume.

5.5 Experiment 4: Real Date Set

In experiment 4, we evaluate the performance of SSPL on a real data set from some communication corporation. It is a part of communication log. The log is divided into several partitions, each of which consists of 10,000,000 tuples with 72 attributes. The length of each tuple is 376 bytes. We select two attributes with the largest variance to be skyline criteria. Five partitions are utilized in experiment 4. We fix skyline criteria size 2 and evaluate SSPL with varying data volumes. Also, RPCIT is preconstructed for SSPL. As shown in Fig. 9a, SSPL runs 111.8 times faster than LESS. And SSPL runs faster than ZSearch and slower than SUBSKY. Fig. 9b depicts the decomposition of SSPL. Although phase 2 still takes majority of execution time of SSPL, the proportion taken by loading EGBFT is higher than those in experiment 1 to 3. The reason is that the sequential disk seek is faster in a smaller data file. As displayed in Fig. 9c, SSPL retrieves 164839.9 times fewer tuples than LESS, 279.05 times fewer tuples than ZSearch and 10.6 times fewer tuples than SUBSKY. Fig. 9d shows scan depth of SSPL, which increases with larger data volume. As shown in Fig. 9e, the pruning ratio of early pruning first decreases when data volume increases from 3.76G to 7.52G. Then the pruning ratio increases with larger data volume. The decline is because the positional index of EGBFT in 3.76G is 17 while the positional index in 7.52G increases to 19. And as the data volume becomes larger, the used positional indexes of EGBFT do not change. The pruning effect of late pruning is illustrated in Fig. 9f.

5.6 Summary

In the experiments, compared to LESS, SSPL runs up to three orders of magnitude faster and retrieves up to six orders of magnitude fewer tuples with the help of sorted positional index lists and pruning operation. It reflects the value of SSPL on processing skyline on big data. In uniform distribution, the experimental results verify the theoretical analysis in the paper. It is shown that SSPL performs well when the size of skyline criteria is small. Especially if the attributes are positively correlated, the performance of SSPL is quite satisfactory.

We also evaluate SSPL against tree-based algorithms (ZSearch and SUBSKY). The experimental results show that the performance of SSPL can be comparable with tree-based algorithms. It should be noted that tree-based algorithms actually require constructing exponential number of indexes to cover the required skyline criteria, which is rather difficult (if not impossible). In the experiment, we sort the table according to the used skyline criteria and in practice, the application of tree-based algorithms is seriously limited. Basically, SSPL utilizes much smaller and more practical data-structures to achieve a performance comparable to tree-based algorithms.

As shown in Fig. 7a, when the size of skyline criteria increases, the skyline computation times of SSPL, ZSearch,

and SUBSKY increase quickly, which can be treated as the effect of "Curse of dimensionality" [33]. It seems that it is not promising for applying SSPL in a real application. However, SSPL retrieves up to six orders of magnitude fewer tuples than LESS (48 times fewer tuples than LESS when the size of skyline criteria is 5), and a large proportion of execution time of SSPL in our experiment is consumed by rotational latency time and seek time in mechanical disk used here. It can be expected that SSPL will have a much better performance when performed on solid state disk, which do not spin, or seek, and will show huge superiority in random access operation. Although solid state disk is much more expensive than mechanical disk currently, it will come down in price greatly and be used extensively in the near future. Besides, in practical application, the size of skyline criteria is typically small, which indicates the extensive practicality of SSPL.

At first glance, it seems that the performance comparison between SSPL and LESS is unfair. After all, SSPL preconstructs sorted positional index lists for its processing, while LESS does not assume the preprocessing. However, it is noted that the algorithm for solving a problem is developed under the consideration of a tradeoff among its implementation complexity, space requirement and performance. Although LESS does not require any preprocessing, its performance on big data is poor. By utilizing the preconstructed data structures, SSPL achieves a much better performance. It is known that the preconstructed data structures can be used repeatedly until the next update, and big data update is performed in batch and periodic mode. Therefore, the preprocessing procedures and the data-structures can be treated as a graceful tradeoff with low preconstruction cost and small space requirement for a much higher performance.

6 Conclusion

In this paper, we consider the problem of processing skyline query on big data. It is analyzed that the current skyline algorithms cannot perform skyline on big data efficiently. This paper proposes a novel skyline algorithm SSPL, which utilizes sorted positional index lists of low space overhead, to reduce the I/O cost significantly. SSPL consists of two phases. In phase 1, it retrieves the sorted positional index lists specified by skyline criteria in a round-robin fashion until there is a candidate positional index seen in all of the involved lists. In phase 2, SSPL performs a sequential and selective scan on the table by the candidate positional indexes obtained in phase 1 to compute skyline results. Early pruning and late pruning are presented in the paper to discard the unsatisfied candidate positional indexes in phase 1. The experimental results on synthetic and real data sets show that SSPL has a significant advantage over the existing skyline algorithms.

Acknowledgments

This work was supported in part by the National Basic Research (973) Program of China under grant no. 2012CB316200, the National Science Foundation of China under grant nos. 61190115, 61173022, 61033015, 60831160525, 61272046, and 60903016.

References

[1] I. Bartolini, P. Ciaccia, and M. Patella, "Efficient Sort-Based Skyline Evaluation," *ACM Trans. Database Systems,* vol. 33, no. 4, pp. 31:1-31:49, 2008.

[2] I. Bartolini, Z. Zhang, and D. Papadias, "Collaborative Filtering with Personalized Skylines," *IEEE Trans. Knowledge Data Eng.,* vol. 23, no. 2, pp. 190-203, Feb. 2011.

[3] J.L. Bentley, H.T. Kung, M. Schkolnick, and C.D. Thompson, "On the Average Number of Maxima in a Set of Vectors and Applications," *J. ACM,* vol. 25, no. 4, pp. 536-543, 1978.

[4] J.L. Bentley, K.L. Clarkson, and D.B. Levine, "Fast Linear Expected-Time Algorithms for Computing Maxima and Convex Hulls," *Proc. First Ann. ACM-SIAM Symp. Discrete Algorithms (SODA '90),* pp. 179-187, 1990.

[5] M. Berger, *Geometry I,* vol. 1, Springer, 1987.

[6] B.H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Comm. ACM,* vol. 13, no. 7, pp. 422-426, 1970.

[7] S. Börzsönyi, D. Kossmann, and K. Stocker, "The Skyline Operator," *Proc. 17th Int'l Conf. Data Eng. (ICDE '01),* pp. 421-430, 2001.

[8] R.E. Bryan, "Data-Intensive Supercomputing: The Case for Disc," Technical Report CMU-CS-07-128, School of Computer Science, Carnegie Mellon Univ., 2007.

[9] C.-Y. Chan, H.V. Jagadish, K.-L. Tan, A.K.H. Tung, and Z. Zhang, "Finding K-Dominant Skylines in High Dimensional Space," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '06),* pp. 503-514, 2006.

[10] L. Chen and X. Lian, "Efficient Processing of Metric Skyline Queries," *IEEE Trans. Knowledge Data Eng.,* vol. 21, no. 3, pp. 351-365, Mar. 2009.

[11] L. Chen, B. Cui, and H. Lu, "Constrained Skyline Query Processing against Distributed Data Sites," *IEEE Trans. Knowledge Data Eng.,* vol. 23, no. 2, pp. 204-217, Feb. 2011.

[12] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with Presorting," *Proc. 19th Int'l Conf. Data Eng. (ICDE '03),* pp. 717-719, 2003.

[13] B. Cui, L. Chen, L. Xu, H. Lu, G. Song, and Q. Xu, "Efficient Skyline Computation in Structured Peer-to-Peer Systems," *IEEE Trans. Knowledge Data Eng.,* vol. 21, no. 7, pp. 1059-1072, July 2009.

[14] R. Fagin, A. Lotem, and M. Naor, "Optimal Aggregation Algorithms for Middleware," *Proc. 31st ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS '01),* pp. 102-113, 2001.

[15] W. Feller, *An Introduction to Probability Theory and Its Applications,* vol. 1, third ed. Wiley, 1968.

[16] D. Freedman, R. Pisani, and R. Purves, *Statistics,* fourth ed. W.W. Norton & Company, 2007.

[17] M. Gibas, G. Canahuate, and H. Ferhatosmanoglu, "Online Index Recommendations for High-Dimensional Databases Using Query Workloads," *IEEE Trans. Knowledge and Data Eng.,* vol. 20, no. 2, pp. 246-260, Feb. 2008.

[18] P. Godfrey, "Skyline Cardinality for Relational Processing," *Foundations of Information and Knowledge Systems,* vol. 2942, pp. 78-97, Springer Berlin/Heidelberg, 2004.

[19] P. Godfrey, R. Shipley, and J. Gryz, "Algorithms and Analyses for Maximal Vector Computation," *The VLDB J.,* vol. 16, no. 1, pp. 5-28, 2007.

[20] J. Gray and P.J. Shenoy, "Rules of Thumb in Data Engineering," *Proc. 16th Int'l Conf. Data Eng. (ICDE '00),* pp. 3-12, 2000.

[21] K. Hose and A. Vlachou, "A Survey of Skyline Processing in Highly Distributed Environments," *The VLDB J.,* vol. 21, no. 3, pp. 359-384, 2012.

[22] D. Kossmann, F. Ramsak, and S. Rost, "Shooting Stars in the Sky: An Online Algorithm for Skyline Queries," *Proc. 28th Int'l Conf. Very Large Data Bases (VLDB '02),* pp. 275-286, 2002.

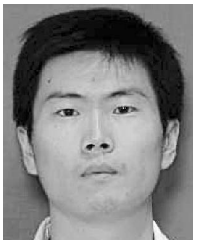
[23] H.T. Kung, F. Luccio, and F.P. Preparata, "On Finding the Maxima of a Set of Vectors," *J. ACM,* vol. 22, no. 4, pp. 469-476, 1975.

[24] K.C. Lee, W.-C. Lee, B. Zheng, H. Li, and Y. Tian, "Z-Sky: An Efficient Skyline Query Processing Framework Based on Z-Order," *The VLDB J.,* vol. 19, no. 3, pp. 333-362, 2010.

[25] N. Mamoulis, M.L. Yiu, K.H. Cheng, and D.W. Cheung, "Efficient Top-K Aggregation of Ranked Inputs," *ACM Trans. Database Systems,* vol. 32, no. 3, p. 19, 2007.

[26] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive Skyline Computation in Database Systems," *ACM Trans. Database Systems,* vol. 30, no. 1, pp. 41-82, 2005.

[27] Seagate, "Barracuda xt: No Compromise. Speed and Capacity for High-Performance Desktop Systems," http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_xt.pdf, 2012.
[28] C. Sheng and Y. Tao, "On Finding Skylines in External Memory," *Proc. 30th ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS '11)*, pp. 107-116, 2011.
[29] M. Stonebraker et al., "C-Store: A Column-Oriented DBMS," *Proc. 31st Int'l Conf. Very Large Data Bases (VLDB '05)*, pp. 553-564, 2005.
[30] K.-L. Tan, P.-K. Eng, and B.C. Ooi, "Efficient Progressive Skyline Computation," *Proc. 27th Int'l Conf. Very Large Data Bases (VLDB '01)*, pp. 301-310, 2001.
[31] Y. Tao, X. Xiao, and J. Pei, "Efficient Skyline and Top-K Retrieval in Subspaces," *IEEE Trans. Knowledge Data Eng.*, vol. 19, no. 8, pp. 1072-1088, Aug. 2007.
[32] Tom's Hardware, "Hard Drives: 40 mb to 750 gb," http://www.tomshardware.com/reviews/15-years-of-hard-drive-history,1368-2.html, 2006.
[33] Wikipedia, "Curse of Dimensionality,"http://en.wikipedia.org/wiki/Curse_of_dimensionality, 2012.
[34] Wikipedia, "Equality of Arithmetic and Geometric Means," http://en.wikipedia.org/wiki/Inequality_of_arithmetic_and_geometric_means, 2012.
[35] H.K.T. Wong, J. Li, F. Olken, D. Rotem, and L. Wong, "Bit Transposition for Very Large Scientific and Statistical Databases," *Algorithmica*, vol. 1, no. 3, pp. 289-309, 1986.
[36] Z. Zhang, H. Lu, B.C. Ooi, and A.K. Tung, "Understanding the Meaning of a Shifted Sky: A General Framework on Extending Skyline Query," *The VLDB J.*, vol. 19, no. 2, pp. 181-201, 2010.
[37] L. Zhu, Y. Tao, and S. Zhou, "Distributed Skyline Retrieval with Low Bandwidth Consumption," *IEEE Trans. Knowledge Data Eng.*, vol. 21, no. 3, pp. 384-400, Mar. 2009.



Xixian Han received the master's and PhD degrees from the School of Computer Science and Technology, Harbin Institute of Technology, in 2006 and 2012, respectively. He is a lecturer in the School of Computer Science and Technology, Harbin Institute of Technology, China. His main research interests include massive data management and data-intensive computing.

Jianzhong Li is a chair of the Department of Computer Science and Engineering at Harbin Institute of Technology, China. He is also a professor in the School of Computer Science and Technology, the dean of the School of Computer Science and Technology, and the dean of the School of Software at Heilongjiang University, China. His current research interests include data-intensive computing, wireless sensor networks, and CPS. He is a member of the IEEE.

Donghua Yang received the master's and PhD degrees from the School of Computer Science and Technology, Harbin Institute of Technology, in 2003 and 2008, respectively. He is a lecturer in the Center for High Performance Computing, The Academy of Fundamental and Interdisciplinary Sciences, Harbin Institute of Technology, China. His research interest includes database, massive data management, query processing, and cloud computing.

Jinbao Wang received the BE and ME degrees in computer science and technology from the Harbin Institute of Technology, in 2006 and 2008, respectively. Currently, he is working toward the PhD degree in the School of Computer Science and Technology at Harbin Institute of Technology. His main research interests include cloud-computing and data-intensive computing.