Data Science and Engineering (2022) 7:102–119
https://doi.org/10.1007/s41019-022-00183-7

RESEARCH PAPERS

Efficient Skyline Computation on Massive Incomplete Data

Jingxuan He[1] · Xixian Han[1]

Received: 6 March 2021 / Revised: 9 February 2022 / Accepted: 9 March 2022 / Published online: 3 April 2022
© The Author(s) 2022

Abstract
Incomplete skyline query is an important operation to filter out pareto-optimal tuples on incomplete data. It is harder than skyline due to intransitivity and cyclic dominance. It is analyzed that the existing algorithms cannot process incomplete skyline on massive data efficiently. This paper proposes a novel table-scan-based TSI algorithm to deal with incomplete skyline on massive data with high efficiency. TSI algorithm solves the issues of intransitivity and cyclic dominance by two separate stages. In stage 1, TSI computes the candidates by a sequential scan on the table. The tuples dominated by others are discarded directly in stage 1. In stage 2, TSI refines the candidates by another sequential scan. The pruning operation is devised in this paper to reduce the execution cost of TSI. By the assistant structures, TSI can skip majority of the tuples in phase 1 without retrieving it actually. The extensive experimental results, which are conducted on synthetic and real-life data sets, show that TSI can compute skyline on massive incomplete data efficiently.

Keywords Massive data · TSI · Incomplete skyline · Pruning operation

1 Introduction

The skyline operator filters out a set of interesting tuples from a potential huge data set. Among the specified skyline criteria, a tuple p is said to dominate another tuple q if p is strictly better than q in at least one attribute, and no worse than q in the other attributes. The skyline query actually discovers all tuples which are not dominated by any other tuples.

Due to its practical importance, skyline queries have received extensive attentions [2, 3, 5, 6, 8, 9, 14–17, 19, 20]. However, the overwhelming majority of the existing algorithms only consider the data set of complete attributes, i.e., all the attributes of every tuple are available. In real-life applications, because of the reasons such as the delivery failure or the deliberate concealment, the data set we encounter often is incomplete, i.e., some attributes of tuples are unknown [13]. On incomplete data, the existing skyline algorithms cannot be applied directly, since all of them assume the transitivity of dominance relationship. On complete data, the transitivity rule is that: if p_1 dominates p_2, and p_2 dominates p_3, obviously p_1 dominates p_3 by the definition of dominance. The transitivity is the basis of the efficiency of the existing skyline algorithms which utilize indexing, partitioning and pre-sorting operation. On incomplete data, some attributes of tuples are missing, the traditional definition of dominance does not hold any more, and the dominance relationship is re-defined on incomplete data. Given the skyline criteria, p and q are two tuples on incomplete data, let C be the common complete attributes of p and q among the skyline criteria, p dominates q if p is no worse than q among C and strictly better than q in at least one attribute among C. From the dominance relationship defined above, transitivity does not hold on incomplete data.

As illustrated in Fig. 1, the specified skyline criteria are $\{A_1, A_2, A_3\}$. In the table, p_1 dominates p_2 since the common attribute among the skyline criteria of p_1 and p_2 is A_1, $p_1.A_1 < p_2.A_1$. Similarly, p_2 dominates p_3. But p_1 does not dominate p_3 here and transitivity does not hold. Besides, it is found that p_3 dominates p_1. On incomplete data, we may face the issue of cyclic dominance. The two issues, intransitivity and cyclic dominance, make the processing of skyline on incomplete data different from the skyline on complete data.

The current incomplete skyline algorithms can be classified into three categories: replace-based algorithms [7],

✉ Xixian Han
xxhan1981@163.com

1 School of Computer Science and Technology, Harbin Institute of Technology, No.92, Xidazhi Street, Harbin, Heilongjiang, China

	A_1	A_2	A_3	A_4
p_1	1	4	–	6
p_2	2	–	3	1
p_3	–	2	5	3

Fig. 1 Data set of example table

sorted-based algorithms [1], and bucket-based algorithms [7, 10]. The replace-based algorithms first replace the incomplete attributes with a specific value, then compute traditional skyline on transformed data, and finally refine the candidate to compute the results by pairwise comparison. Normally, the number of candidate on massive data is large and the pairwise comparison is significantly expensive. Sorted-based algorithms utilize the selected tuples with possible high dominance via pre-sorted structures one by one to prune the non-skyline tuples. It usually performs many passes on scan on the table and will incur high I/O cost on massive data. Bucket-based algorithms first split the tuples into different buckets according to their attribute encoding to make the tuples in the same buckets have the same encoding and hold the transitivity rule, then compute local skyline results on every buckets, and finally merge the local skyline results to obtain the results. In incomplete skyline computation, the skyline criteria size usually is greater than that on complete data due to the cyclic dominance, the bucket number involved in bucket-based algorithms often is large, and the number of local skyline results is relatively great. The computation operation and merge operation of local skyline results often incur high computation cost and I/O cost on massive data. To sum up, the existing algorithms cannot process incomplete skyline query on massive data efficiently.

Based on the discussion above, this paper proposes TSI algorithm (*T*able-scan-based *S*kyline over *I*ncomplete data) to compute skyline results on massive incomplete data with high efficiency. In order to reduce the computation cost and I/O cost, the execution of TSI consists of two stages. In stage 1, TSI performs a sequential scan on the table and maintains candidates in memory. For each tuple t retrieved currently in stage 1, any candidate dominated by t is removed. And if t is not dominated by any candidates, t is added to the candidate set. In stage 1, TSI does not consider the intransitivity and cyclic dominance in incomplete skyline computation, but just discards the tuples which are not final results definitely. In stage 2, another sequential scan is executed to refine the candidates. For each tuple t retrieved currently in stage 2, it discards any candidates dominated by it. When stage 2 terminates, the candidates in memory are the incomplete skyline results. In this paper, it is found that the cost in stage 1 dominates the overall cost of TSI, so a pruning operation is devised to skip the tuples in stage 1. The useful data structures are pre-constructed, which is used to check whether a tuple is dominated before retrieving it. The extensive experiments are conducted on synthetic and real-life data sets. The experimental results show that the pruning operation can skip overwhelming majority of tuples in stage 1 and TSI outperforms the existing algorithms significantly.

The contributions of this paper are listed as follows:

- This paper proposes a novel table-scan-based TSI algorithm of two stages to process skyline query on massive incomplete data efficiently.
- Two novel data structure is designed to maintain information of tuples and obtain pruning tuples with strong dominance capability.
- This paper devises efficient pruning operations to reduce the execution cost of TSI, which directly skips the tuples dominated by some tuples before retrieving them actually.
- The experimental results show that TSI can compute incomplete skyline on massive data efficiently.

The rest of the paper is organized as follows. The related work is surveyed in Sect. 2, followed by preliminaries in Sect. 3. The existing algorithms are analyzed in Sect. 4. Baseline algorithm is developed in Sect. 5. Section 6 introduces TSI algorithm. The performance evaluation is provided in Sect. 7. Section 8 concludes the paper.

2 Related Work

Since [2] first introduces the skyline operator into database environment, skyline has been studied extensively by database researchers [2, 3, 5, 6, 8, 11, 14–17, 20]. However, the most of the existing skyline algorithms only consider the complete data, and they utilize the transitivity of dominance relationship to acquire significant pruning power. They cannot be directly used for the skyline query on incomplete data, where the dominance relationship is intransitivity and cyclic. In the rest of this section, we survey the skyline algorithms on incomplete data. The current incomplete skyline algorithms can be classified into three categories: replace-based algorithms, sorted-based algorithms, and bucket-based algorithms.

2.1 Replace-Based Algorithms

Khalefa et al. [7] propose a set of skyline algorithms for incomplete data. The first two algorithms, replacement and bucket, are the extension of the existing skyline algorithms to accommodate the incomplete data. Replacement algorithm first replaces the incomplete attributes by a special value to transform the incomplete data to complete data. Traditional skyline algorithm can be used to compute the skyline results SKY_{comp} on the transformed complete data, which also is the superset of the skyline results on the incomplete data. Finally, the tuples in SKY_{comp} are transformed into their original incomplete form, and the exhaustive pairwise comparison between all tuples in SKY_{comp} is performed to compute the final results. Bucket algorithm first divides all the tuples on incomplete data into different buckets to make all tuples in the same bucket have the same bitmap representation. The dominance relationship within the same bucket is transitive now since the tuples here have the same bitmap representation. The traditional skyline algorithm is utilized to compute the skyline results within each bucket, which is called local skyline. The local skyline results for all buckets are merged as the candidate skyline. The exhaustive pairwise comparison is performed on the candidate skyline to compute the query answer. ISkyline algorithm employs two new concepts, virtual points and shadow skylines, to improve bucket algorithm. The execution of ISkyline consists of three phases. In phase I, each newly retrieved tuple is compared against the local skyline and the virtual points to determine whether the tuple needs to be (a) stored in the local skyline, (b) stored in the shadow skyline, (c) discarded directly. In phase II, the tuples newly inserted into local skyline are compared with the current candidate skyline; ISkyline updates the candidate skyline and the virtual points correspondingly. Every time t tuples are kept in the candidate skyline, ISkyline enters into phase III, updates the global skyline, and clears current candidate skyline. The similar processing continues until the end of input is reached and ISkyline returns the global skyline.

The replace-based algorithms first replace the incomplete attributes with a specific value, then compute traditional skyline on transformed data, and finally refine the candidate to compute the results by pairwise comparison. Normally, the number of candidate on massive data is large and the pairwise comparison is significantly expensive.

2.2 Sorted-Based Algorithms

Bharuka et al. [1] propose a sort-based skyline algorithm SIDS to evaluate the skyline over incomplete data. SIDS first sorts the incomplete data D in non-descending order for each attribute. Let D_i be the sorted list with respect to the ith attribute. Only the ids of the tuples are kept in D_i, and the ids of the tuples whose ith attributes are incomplete are not stored. SIDS performs a round-robin retrieval on the sorted lists. For each retrieved data p, if it is not retrieved before, it is compared with each data q in the candidate set, which is initialized to be the whole incomplete data. If p and q are compared already, the next data in the candidate set are retrieved and processed. Otherwise, if p dominates q, q is removed from the candidate set. And if q dominates p and p is in candidate set, p is removed from the candidate set also. If the number of p being retrieved during the round-robin retrieval is equal to the number of its complete attributes and p is not pruned yet, p can be reported to be one of the skyline results. SIDS terminates when candidate set becomes empty or all points in sorted lists are processed at least once.

Sorted-based algorithms utilize the selected tuples with possible high dominance via pre-sorted structures one by one to prune the non-skyline tuples. It usually performs many passes on scan on the table and will incur high I/O cost on massive data.

2.3 Bucket-Based Algorithms

Lee et al. [10] propose a sorting-based SOBA algorithm to optimize the bucket algorithm. Similar to the bucket algorithm, SOBA also first divides the incomplete data into a set of buckets according to their bitmap representation, then computes the local skyline of tuples in each bucket, and finally performs the pairwise comparison for the skyline candidates (the collection of all local skylines). SOBA uses two techniques to reduce the dominance tests for the skyline candidates. The first technique is to sort the buckets in ascending order of the decimal numbers of the bitmap representation. This can identify the non-skyline points as early as possible. The second technique is to rearrange the order of tuples within the bucket. By sorting tuples in the ascending order of the sum of the complete attributes, the tuples accessed earlier have the higher probability to dominate other tuples and this can help reduce the number of dominance tests further.

Bucket-based algorithms first split the tuples into different buckets according to their attribute encoding to make the tuples in the same buckets have the same encoding and hold the transitivity rule, then compute local skyline results on every buckets, and finally merge the local skyline results to obtain the results. In incomplete skyline computation, the skyline criteria size usually is greater than that on complete data due to the cyclic dominance, the bucket number involved in bucket-based algorithms often is large, and the number of local skyline results is relatively great. The computation operation and merge operation of local skyline

Table 1 Symbols description

Symbol	Description
T	An incomplete table
T_{part}	The current part of T loaded in memory
t	A tuple in T
C	The common complete attribute(s)
PI	The positional index of tuple X
S	The size of allocated memory for storing tuples of T in each time X
S_{cnd}	A set maintaining candidate tuples
SL_i	The sorted list which is built for the i-th attribute
MCR	The bit-vector representing the membership checking result of SL_i
RIA	The bit-vector representing whether the attribute is complete
S_c	The set of the complete attributes of t
NUM_c	The number of the complete attributes for each tuple

results often incur high computation cost and I/O cost on massive data.

For the algorithms mentioned above, the dominance over incomplete data is defined on the common complete attributes. There are also other definitions of dominance over incomplete data. Zhang et al. [21] propose a general framework to extend skyline query. For each attribute, they first retrieve the probability distribution function of the values in the attribute by all the non-missing values on the attribute and then convert incomplete tuples to complete data by estimating all missing attributes. And a mapping dominance is defined on the converted data. Zhang et al. [18] propose PISkyline to compute probabilistic skyline on incomplete data. It is considered in [18] that each missing attribute value can be described by a probability density function. The probability is used to measure the preference condition between missing values and the valid values. Then, the probability of a tuple being skyline can be computed. PISkyline returns the K tuples with the highest skyline probability.

Discussion Throughout this paper, we use the definition of dominance over incomplete data as [7]. Firstly, this dominance notion is commonly used in most skyline algorithms over incomplete data. Secondly, the estimation of the incomplete attribute values may be undesirable in some cases. Therefore, we do not guess the incomplete attribute in this paper and not consider such algorithms anymore.

In this paper, we consider the skyline over massive incomplete data, i.e., the data set cannot be kept in memory entirely. It is found that the existing algorithms, including [1, 7, 10], all assume their processing of the in-memory data. Their performance will be seriously degraded on massive data. Since the cardinality of skyline query increases exponentially with respect to the size of skyline criteria [4], replacement algorithm often generates a large number of skyline candidates and the pairwise comparison among the candidates incurs a prohibitively expensive cost. Bucket-based algorithms, such as bucket algorithm, ISkyline and SOBA, have the problem that they have to divide the data set into different buckets. Given the size m of the skyline criteria, the number of the buckets can be as high as $2^m - 1$; this will cause serious performance issue when m is not small. For SIDS, it utilizes one selected tuple to prune the non-skyline tuples in the candidate set, and this incurs a pass of sequential scan on the data. Thus, it requires many passes of scan on the data to finish its execution, and this will incur a high I/O cost on massive data.

3 Preliminaries

Given an incomplete table T of n tuples with attributes $A_1, \dots, A_M$, some attributes of the tuples in T are incomplete. The attributes in T are considered to be numerical type, let $A_1, \dots, A_m$ be the specified skyline criteria. Throughout the paper, it is assumed that the smaller attribute values are preferred. In this paper, the attributes with known values are called *complete attributes*, while the attributes with unknown values are called *incomplete attributes*. $\forall t \in T$, t has at least one complete attribute among $A_1, A_2, \dots, A_m$, while all other attributes have a probability p $(0 < p \leq 1)$ of being incomplete. The frequently used symbols in this paper are listed in Table 1.

The dominance over incomplete data is given in Definition 1. The incomplete skyline returns the tuples in T which are not dominated by any other tuples.

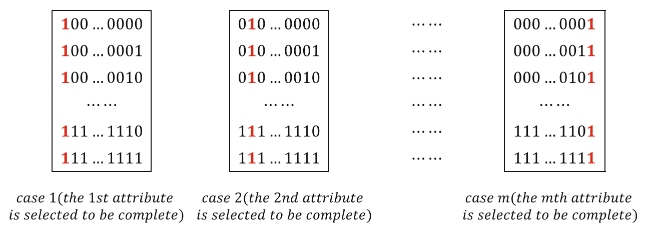


Fig. 2 Different cases of bitmap encoding

Definition 1 (*Dominance over incomplete data*) Given table T and skyline criteria $A_1, \ldots, A_m$, $\forall t_1, t_2 \in T$, let C be their common complete attributes among skyline criteria, t_1 dominates t_2 (denoted by $t_1 \succ t_2$) if $\forall A \in C, t_1.A \leq t_2.A$, and $\exists A \in C, t_1.A < t_2.A$.

Definition 2 (*Positional index*) $\forall t \in T$, its positional index (PI) is a if t is the ath tuple in T.

The positional index is defined in Definition 2. We denoted by $T(a)$ the tuple with $PI = a$, by $T(a, \ldots, b)(a \leq b)$ the tuples in T whose PIs are between a and b, by $T(a, \ldots, b).A_i$ be the set of attribute A_i in $T(a, \ldots, b)$.

4 The Analysis for the Existing Algorithms

The existing skyline algorithms over incomplete data can be classified into three types: replacement-based algorithm, bucket-based algorithm, and sort-based algorithm. As discussed in Sect. 2, replacement-based algorithm usually generates too many skyline candidates and sort-based algorithm often needs to perform many passes on the table before returning the results. They both incur much high computation cost and I/O cost on massive data. In the following part of this section, we analyze the performance of bucket-based algorithm.

Given table T and the skyline criteria $\{A_1, A_2, \ldots, A_m\}$, $\forall t \in T$, t can be encoded by an m-bit vector $t.B$. $\forall i, 1 \leq i \leq m$, if $t.A_i$ is a complete attribute, the ith bit of $t.B$ is 1 (denoted by $t.B(i) = 1$); otherwise, the ith bit of $t.B$ is 0 ($t.B(i) = 0$). Note that the most significant bit is the first bit. Bucket-based algorithm divides tuples in T according to their encoded vectors. Therefore, the tuples in the same bucket share the same vectors, and the transitive dominance relation holds among the tuples in a bucket. Traditional skyline algorithm can be utilized to compute the local skyline within the bucket. Any tuple t_1 dominated by a tuple t_2 in the same bucket can be discarded directly, since it cannot be skyline result and any tuples which can be dominated by t_1 can be dominated by t_2 naturally. Of course, there are other techniques to optimize the pairwise comparison among skyline candidates [10].

As illustrated in Fig. 2, for analysis, we assume that the bitmap encoding of the buckets consists of m cases with equal likelihood: $\exists i, 1 \leq i \leq m$, the values of A_i must be known and other attributes can be unknown with the probability p independently. Given an m-bit $b = (b_1 b_2 \ldots b_m)$ of some bucket, $Cnt1(b) = r$, where $Cnt1$ is a function to return the number of bit 1 in a bit-vector. Of course, in this paper, $1 \leq r \leq m$. The bit-vector b can be occurred in r cases. In each case, the probability of generating b is $(1-p)^{r-1} \times p^{m-r}$, i.e., besides the selected complete attribute, there are $(r-1)$ complete attributes and $(m-r)$ incomplete attributes. Therefore, the probability pr_b of generating b among the overall cases is $pr_b = \frac{r}{m} \times (1-p)^{r-1} \times p^{m-r}$. The number N_b of tuples which have the encoded bit-vector b in T is $n_b = n \times pr_b$.

Theoretically, bucket-based algorithm can split T into all $2^m - 1$ buckets. The size of skyline criteria of skyline on incomplete data usually is greater than that on complete data due to the cyclic dominance. This can be verified in the existing skyline algorithms on incomplete data [1, 7, 10]. Then, the number of all buckets is not small. For example, given $m = 20$, there are possibly 1048575 buckets. Then, bucket-based algorithm has to maintain a large number of buckets. For one thing, this increases the management burden of the file system; for another, this makes each bucket maintain a relatively small number of tuples with not small skyline criteria size.

The size of the skyline candidates for pairwise comparison, i.e., the local skylines of all buckets, is $size_{sc} = \sum_{b=(00\ldots01)}^{(11\ldots11)} |SKY_b|$, where SKY_b are the skyline tuples in bucket b. Under the independent assumption, the number of local skyline in the bucket of encoded bit-vector b can be estimated as $\frac{((\ln n_b)+\gamma)^{r-1}}{(r-1)!}$ [4], where $\gamma \approx 0.57721$ is the Euler-Mascheroni constant. But in this paper, it is found that the cardinality estimation is much lower than the actual cardinality when m is relatively large. Of course, we can use other cardinality estimation methods [12, 22] in such case. For

PI	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A_1	–	71	48	–	56	56	67	26	78	73	65	20	47	86	39	90
A_2	91	–	–	82	–	3	94	20	22	50	–	71	–	6	–	79
A_3	46	51	–	–	80	0	–	38	28	–	87	54	78	–	21	8

Fig. 3 Illustration of row table in running example

The first iteration

candidates in the memory	*operation*
{$T(1)$, $T(2)$, $T(3)$, $T(4)$, $T(5)$, $T(6)$, $T(7)$, $T(8)$}	$T(1)$ *from disk* *dominates* $T(2)$, $T(5)$, $T(7)$ *in memory*
{$T(1)$, ~~$T(2)$~~, $T(3)$, $T(4)$, ~~$T(5)$~~, $T(6)$, ~~$T(7)$~~, $T(8)$}	$T(3)$ *from disk* *dominates* $T(6)$ *in memory*
{$T(1)$, ~~$T(2)$~~, $T(3)$, $T(4)$, ~~$T(5)$~~, ~~$T(6)$~~, ~~$T(7)$~~, $T(8)$}	$T(4)$ *from disk* *dominates* $T(1)$ *in memory*
{~~$T(1)$~~, ~~$T(2)$~~, $T(3)$, $T(4)$, ~~$T(5)$~~, ~~$T(6)$~~, ~~$T(7)$~~, $T(8)$}	$T(6)$ *from disk* *dominates* $T(4)$ *in memory*
{~~$T(1)$~~, ~~$T(2)$~~, $T(3)$, ~~$T(4)$~~, ~~$T(5)$~~, ~~$T(6)$~~, ~~$T(7)$~~, $T(8)$}	$T(8)$ *from disk* *dominates* $T(3)$ *in memory*
{~~$T(1)$~~, ~~$T(2)$~~, ~~$T(3)$~~, ~~$T(4)$~~, ~~$T(5)$~~, ~~$T(6)$~~, ~~$T(7)$~~, $T(8)$}	$T(10)$, $T(11)$, $T(13)$ *from disk* *are dominated by* $T(8)$ *in memory*

The second iteration

candidates in the memory	*operation*
{$T(9)$, $T(12)$, $T(14)$, $T(15)$, $T(16)$}	$T(2)$ *from disk* *dominates* $T(14)$ *in memory*
{$T(9)$, $T(12)$, ~~$T(14)$~~, $T(15)$, $T(16)$}	$T(3)$ *from disk* *dominates* $T(9)$, $T(16)$ *in memory*
{~~$T(9)$~~, $T(12)$, ~~$T(14)$~~, $T(15)$, ~~$T(16)$~~}	$T(4)$ *from disk* *dominates* $T(1)$ *in memory*

Fig. 4 Illustration of execution of BA algorithm

simplicity, we still use the cardinality estimation in [4], since it still can provide useful insight for our analysis. Given $n = 10^8$, $m = 20$ and $p = 0.5$, the total number of all local skyline results is 7641060 even by use of the cardinality formula mentioned above, which is much lower than the actual value. The number of local skyline results, which is used to perform pairwise comparisons, is still too high.

To sum up, the existing skyline algorithms on massive incomplete data all have their performance issue.

5 Baseline Algorithm

The existing algorithms, as mentioned in Secst. 2 and 4, have rather poor performance and very long execution time on massive incomplete data. Therefore, this section first devises a baseline algorithm BA which can be used as a benchmark against the algorithm proposed in this paper. Different from the existing methods, BA adopts a block-nested-loop-like execution. It first retrieves T from the beginning and loads a part of T into the memory, compares the tuples in memory with all tuples in T, removes the dominated tuples in the memory. *Each time the tuples left in memory are compared with all other tuples and can be reported as part of incomplete skyline results*. Then, the next part of T is loaded and the similar processing is executed; the iteration continues until all tuples in T are loaded into memory once and compared with all other tuples. In this paper, let S be the size of allocated memory for storing tuples of T each time, the number of table scan in BA is $\frac{8 \times M \times n}{S} + 1$. In order to reduce the I/O cost in BA, a n-bit bit-vector B_{ret}, each bit initialized with 1 is maintained. In the first iteration, the tuples of size S bytes are loaded into memory. Let T_{part} be the current part of T loaded in memory. The tuples in T_{part} are compared with all tuples in T. $\forall t = T(a)$, if t is dominated by some tuple in T_{part}, the ath bit in B_{ret} is set to 0. Then, in the next iteration, suppose that the next retrieved tuple is $T(b)$, if $B_{ret}(b) = 1$, $T(b)$ is retrieved; otherwise, $T(b)$ skips directly since it cannot be a incomplete skyline tuple.

Example 1 In the rest of this paper, we use a running example, as depicted in Fig. 3, to illustrate the execution of algorithms proposed in this paper. In the running example, we set M to be 3, m to be 3, n to be 16 and S to be 256 bytes. The value field of the attribute is [0, 100). According to the parameters, the execution of *BA* divides into two iterations. In the first iteration, $T(1, \dots, 8)$ are loaded into memory. As depicted in Fig. 4, in the first iteration, only $T(8)$ is left and reported as a incomplete tuple. Besides, $T(10)$, $T(11)$, $T(13)$ are dominated by the in-memory candidates in the first iteration and they are skipped in the second iteration. At the end of the second iteration, $T(12)$ and $T(15)$ are left and reported as incomplete skyline tuples. On the whole, the skyline results in the running example are $\{T(8), T(12), T(15)\}$.

6 TSI Algorithm

In this paper, we propose a new algorithm TSI (***T***able-scan-based ***S***kyline over ***I***ncomplete data) to process skyline over massive incomplete data efficiently. TSI performs two passes of scan on the table to compute the skyline results. Section 6.1 describes the basic execution of TSI algorithm. The pruning operation is presented in Sect. 6.2.

6.1 Basic Process

The basic process of TSI consists of two stages. In stage 1, TSI performs the first-pass scan on T to find the candidate tuples, while in the stage 2, TSI scans T again to discard the candidates which are dominated by some tuple. Algorithm 1 is the pseudo-code of the basic process.

Algorithm 1 TSI_basic(T)

```
Input: $T$ is an incomplete table
Output: $S_{cnd}$ a set maintaining the skyline tuples over $T$
1: initialize $S_{cnd} \leftarrow \emptyset$
2: // Stage 1 find the candidate tuples
3: while $T$ has more tuples do
4:     retrieve the next tuple $t$ of $T$;
5:     if $S_{cnd} = \emptyset$ then
6:         $S_{cnd} \leftarrow S_{cnd} \cup t$;
7:     else
8:         while $S_{cnd}$ has more tuples do
9:             retrieve the next tuple $p$ of $S_{cnd}$;
10:            if $p$ is dominated by $t$ then
11:                remove $p$ from $S_{cnd}$;
12:            end if
13:        end while
14:        if $t$ is dominated by $p$ then
15:            discard $t$;
16:        else
17:            $S_{cnd} \leftarrow S_{cnd} \cup t$;
18:        end if
19:    end if
20: end while
21: // Stage 2 discard the candidates which are dominated by some tuples
22: while $T$ has more tuples do
23:     retrieve the next tuple $t$ of $T$;
24:     while $S_{cnd}$ has more tuples do
25:         retrieve the next tuple $can$ of $S_{cnd}$;
26:         if $can$ is dominated by $t$ then
27:             remove $can$ from $S_{cnd}$;
28:         end if
29:     end while
30: end while
31: return $S_{cnd}$;
```

In stage 1, TSI retrieves the tuples in T sequentially and maintains the candidate tuples in a set S_{cnd} (empty initially) (line 1). Let t be the currently retrieved tuple. If S_{cnd} is empty, TSI keeps t in S_{cnd} (line 5-6). Otherwise, S_{cnd} is iterated over, any candidate which is dominated by t is removed from S_{cnd} (line 10-11). At the end of iteration, if t is dominated by some candidate in S_{cnd}, t is discarded (line 14-15); otherwise, TSI keeps t in S_{cnd} (line 16-17). In stage 1, *TSI does not consider the intransitivity and cyclic dominance of skyline on incomplete data*. Any candidates is discarded if it is dominated by some tuple, even though the candidate may dominate the following tuples. In this way, TSI does not need to maintain the dominated tuples and reduces the in-memory maintenance cost significantly. It is proved in Theorem 1 that S_{cnd} contains a superset of the query results at the end of stage 1.

Theorem 1 *When the first-pass scan of TSI is over, S_{cnd} maintains a superset of skyline results over T.*

Proof $\forall t_1 = T(pi_1)$, if t_1 is a skyline tuple, there is no other tuple in T which can dominate t_1. At the end of stage 1, t_1 obviously will be kept in S_{cnd}. If t_1 is not a skyline tuple, and there is another tuple $t_2 = T(pi_2)$ which can dominate t_1. If $pi_1 < pi_2$, t_2 will be retrieved after t_1 and remove t_1 from S_{cnd}. If $pi_1 > pi_2$, t_2 is retrieved before t_1. If t_2 is dominated by some tuple and discarded, t_1 still will be kept in S_{cnd} at the end of stage 1. Q.E.D.

In stage 2, TSI performs another sequential scan on T. Let t be the currently retrieved tuple (line 22-23), any candidates are removed from S_{cnd} if they are dominated by t (line 26-27). It is proved in Theorem 2 that the candidates in S_{cnd} are the skyline results at the end of stage 2.

candidates in the memory	operation
{}	$T(1)$ is inserted to S_{cnd}
$\{T(1)\}$	$T(2)$ from disk is dominated $T(1)$ in memory
$\{T(1), T(3)\}$	$T(3)$ is inserted to S_{cnd}
$\{T(1), T(3)\}$	$T(4)$ from disk dominates $T(1)$ in memory
$\{T(3), T(4)\}$	$T(5)$ from disk is dominated by $T(3)$ in memory
$\{T(3), T(4)\}$	$T(6)$ from disk dominates $T(4)$ in memory $T(6)$ from disk is dominated by $T(3)$ in memory
$\{T(3)\}$	$T(7)$ from disk is dominated by $T(3)$ in memory
$\{T(3)\}$	$T(8)$ from disk dominates $T(3)$ in memory
$\{T(8)\}$	$T(9)$ is inserted to S_{cnd}
$\{T(8), T(9)\}$	$T(10), T(11)$from disk are dominated by $T(8)$ in memory
$\{T(8), T(9)\}$	$T(12)$ is inserted to S_{cnd}
$\{T(8), T(9), T(12)\}$	$T(13)$ is dominated by $T(8)$and $T(12)$
$\{T(8), T(9).T(12)\}$	$T(14)$ is inserted to S_{cnd}
$\{T(8), T(9).T(12), T(14)\}$	$T(15)$ from disk dominates $T(9)$ and $T(14)$ in memory
$\{T(8), T(12), T(15)\}$	$T(16)$ is inserted to S_{cnd}
$\{T(8), T(12), T(15), T(16)\}$	

Fig. 5 Illustration of execution in stage 1 of TSI

candidates in the memory	operation
$\{T(8), T(12), T(15), T(16)\}$	$T(1), T(2)$from disk do not dominate candidate in memory
$\{T(8), T(12), T(15), T(16)\}$	$T(3)$from disk dominates $T(16)$ in memory
$\{T(8), T(12), T(15),$ ~~$T(16)$~~$\}$	$T(4, \ldots, 16)$from disk do not dominate candidates in memory

Fig. 6 Illustration of execution in stage 2 of TSI

Theorem 2 *When the second-pass scan of TSI is over,* S_{cnd} *maintains the skyline results over* T.

Proof $\forall t_1 \in S_{cnd}$, if t_1 is not a skyline tuple, there is another tuple $t_2 = T(pi_2)$ which can dominate t_1. In the second-pass scan, TSI will discard t_1 when retrieving t_2. Q.E.D.

The existing algorithms utilize many methods, such as replacement, sortedness and bucket, to deal with intransitivity and cyclic dominance. They usually incur high execution cost on massive incomplete data, as analyzed in Sects. 2 and 4. In this paper, TSI neglects the intransitivity and cyclic dominance in the first-pass scan and leaves the refinement of the skyline results in the second-pass scan.

Example 2 The execution in stage 1 of TSI in the running example is illustrated in Fig. 5. Initially, the candidate set S_{cnd} is empty. Then, as the first sequential scan is performed, $S_{cnd} = \{T(8), T(12), T(15), T(16)\}$ at the end of stage 1. In stage 2, another sequential scan is executed to refine the candidates. As depicted in Fig. 6, $T(16)$ in S_{cnd} is dominated by $T(3)$. Finally, TSI returns $\{T(8), T(12), T(15)\}$ as incomplete skyline results.

Time complexity On massive incomplete data, the majority of the execution cost of TSI is consumed in stage 1. The reason is that every tuple retrieved in stage 1 needs to compare with all candidates in S_{cnd} and the size of S_{cnd} increases during the first-pass scan on T, while the size of S_{cnd} decreases gradually in stage 2.

Time complexity of stage 1. As shown in Algorithm 1, the time complexity of stage 1 is determined by the nested loop, the outer loop from line 3 to Line 20, and the inner loop from Line 8 to Line 13. Assume that there are n tuples in the incomplete table, in other words, algorithm 1 needs to retrieve n tuples. The iteration count of the outer loop is $O(n)$, since time complexity is the amount of time taken by an algorithm to run as a function of the input size. The inner loop involves one sequential scan on S_{cnd}, whose size is no more than n. For each iteration in the inner loop, the operations take in constant time; thus, the time complexity of the inner loop is $O(|S_{cnd}|)$. On the whole, the time complexity of stage 1 is determined by the number of tuples in T and the number of candidates in S_{cnd}, i.e., the time complexity of stage 1 is $O(n * |S_{cnd}|)$.

Time complexity of stage 2. The execution of stage 2 is described in Algorithm 1. Obviously, the cost of stage 2 is similar to stage 1, i.e., the product of n and the size of S_{cnd}; it might be insignificant compared with the cost of the following operations. The reason is that if the skyline candidates are relatively small, the size of S_{cnd} with skyline subset generating in stage 1 is much large than the size of S_{cnd} with skyline tuples generating in stage 2 and the size of S_{cnd} in stage 1 often dominates the overall execution cost. On the whole, the time complexity of algorithm 1 is $O(n^2)$.

In Sect. 6.2, we will propose pruning method to skip the unnecessary tuples in the sequential scan to improve the performance TSI further.

6.2 Pruning Operation

6.2.1 Intuitive Idea

On massive incomplete data, it is analyzed that the majority of the execution cost of TSI is consumed in stage 1. In stage 1, TSI computes the candidates of the skyline over T. *Obviously, any tuple must not be a skyline tuple if it is dominated by some tuple*. In stage 1, TSI utilizes some pre-constructed data structure to skip the tuples in T which are dominated. In this way, TSI will speed up its execution in stage 1, since the pruning operation not only reduces the I/O cost to retrieve tuples, but also reduces the computation cost of dominance checking.

6.2.2 Dominance Checking on Incomplete Data

Given $t_1 \in T$, $\forall t_2 \in T$, let C be the common complete attributes among skyline criteria of t_1 and t_2. For one thing, if $t_1 \succ t_2$, it means that $\forall A \in C$, $t_1.A \le t_2.A$ and $\exists A \in C$, $t_1.A < t_2.A$. Suppose that t_1 is obtained currently, we can utilize the values of t_1 to skip the tuples dominated by it. For another, it C is empty, t_1 and t_2 cannot be compared in terms of dominance checking. Therefore, the key to the dominance checking on incomplete data is (1) the comparison of complete attributes, (2) the representation of incomplete attributes. In the following, we introduce how to construct data structures to solve the two issues.

In the paper, the value of any incomplete attribute is regarded as the positive infinity since the smaller values are preferred. Given table $T(A_1, \dots, A_M)$, the sorted list $SL_i(1 \le i \le M)$ is built for each attribute. The schema of SL_i is $SL_i(PI_T, A_i)$, where PI_T is the positional index of the tuple in T, and the tuples of SL_i are arranged in the ascending order of A_i. By the sorted lists, TSI constructs the structure MCR (*M*embership *C*hecking *R*esult) to compare the complete attributes. For sorted list $SL_i(1 \le i \le M)$, $MCR_{i,b}(1 \le b \le \lfloor \log_2 n \rfloor)$ is a n-bit bit-vector representing the membership checking results of $SL_i(1, \dots, 2^b).PI_T$. $\forall t = T(a)(1 \le a \le n)$, if $a \in SL_i(1, \dots, 2^b).PI_T$, $MCR_{i,b}(a) = 1$; otherwise, $MCR_{i,b}(a) = 0$. $MCR_{i,b}(a)$ is the ath bit of $MCR_{i,b}$. The maximum values of $SL_i(1, \dots, 2^b).A_i$ $(1 \le b \le \lfloor \log_2 n \rfloor)$ are kept in a array ITV_i, i.e., $ITV_i[b] = SL_i(2^b).A_i$.

For the representation of incomplete attributes, TSI performs a sequential scan on T and constructs the structure RIA, which consists of M n-bit bit-vectors. For $RIA_i(1 \le i \le M)$, $\forall t = T(a)(1 \le a \le n)$, if $T(a).A_i$ is a complete attribute $RIA_i(a) = 1$; otherwise, $RIA_i(a) = 0$.

Example 3 The required data structures mentioned above are illustrated in Fig. 7. SL_1, SL_2, SL_3 are three sorted lists, whose elements are arranged in the ascending order of A_1, A_2, A_3, respectively. $MCR_{1,1}$ is a 16-bit bit-vector representing the membership checking results of $SL_1(1, 2^1).PI_T$, i.e., 12 and 8. Therefore, the 8th bit and 12th bit in $MCR_{1,1}$ are 1, $MCR_{1,1} = 0000000100010000$. ITV_1 keeps the attribute values of exponential gaps in SL_1, i.e., $SL_1(2^1).A_1$, $SL_1(2^2).A_1$, $SL_1(2^3).A_1$, $SL_1(2^4).A_1$, $ITV_1 = \{26, 47, 65, +\infty\}$. The other MCR bit-vectors and other ITVs can be obtained similarly. The structure RIA_i represents the incomplete values of A_i.

	PI	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SL_1	PI_T	12	8	15	13	3	5	6	11	7	2	10	9	14	16	1	4
	A_1	20	26	37	47	48	56	56	65	67	71	73	78	86	90	∞	∞
SL_2	PI_T	6	14	8	9	10	12	16	4	1	7	11	2	13	3	15	5
	A_2	3	6	20	22	50	71	78	82	91	94	∞	∞	∞	∞	∞	∞
SL_3	PI_T	6	16	15	9	8	1	2	12	13	5	11	3	10	4	14	7
	A_3	0	8	21	28	38	46	51	54	78	80	87	∞	∞	∞	∞	∞

$MCR_{1,1} = 0000000100010000$ $\quad$ $MCR_{1,2} = 0000000100011010$
$MCR_{1,3} = 0010110100111010$ $\quad$ $MCR_{1,4} = 1111111111111111$

$MCR_{2,1} = 0000010000000100$ $\quad$ $MCR_{2,2} = 0000010110000100$
$MCR_{2,3} = 0001010111010101$ $\quad$ $MCR_{2,4} = 1111111111111111$

$MCR_{3,1} = 0000010000000001$ $\quad$ $MCR_{3,2} = 0000010010000011$
$MCR_{3,3} = 1100010110010011$ $\quad$ $MCR_{3,4} = 1111111111111111$

$ITV_1 = \{26,47,65,\infty\}$ $\quad$ $ITV_2 = \{6,22,82,\infty\}$ $\quad$ $ITV_3 = \{8,28,54,\infty\}$

$RIA_1 = 0110111111111111$ $\quad$ $RIA_2 = 1001011111010101$
$RIA_3 = 1100110110111011$

Fig. 7 Illustration of MCR and RIA in the running example

In the running example, $T(1).A_1$ and $T(4).A_1$ are incomplete attribute values, therefore, $RIA_1 = 0110111111111111$. Similarly, we can generate RIA_2 and RIA_3.

By the structures MCR and RIA, given $t_1 \in T$, we want to know which tuples in T are dominated by t_1. Let S_c be set of the complete attributes among $A_1, A_2, \dots, A_m$ of t_1, without loss of generality, assume that $S_c = \{A_1, \dots, A_{|S_c|}\}$. $\forall A_i \in S_c(1 \le i \le |S_c|)$, we determine the first value $ITV_i[b_i]$ of ITV_i which is greater than $t_1.A_i$, i.e., $ITV_i[b_i - 1] \le t_1.A_i < ITV_i[b_i]$, here $ITV_i[0]$ is assigned negative infinity. Let DBV_{t_1} be the n-bit bit-vector of dominance checking corresponding to t_1, whose bits are initialized to bit 1. It is proved by Theorem 3 that the bit 1s of $DBV_{t_1} = (\bigwedge_{i=1}^{|S_c|} \neg MCR_{i,b_i}) \land (\bigvee_{i=1}^{|S_c|} RIA_i)$ correspond to the tuples dominated by t_1.

Theorem 3 *The bit 1s of* $DBV_{t_1} = (\bigwedge_{i=1}^{|S_c|} \neg MCR_{i,b_i}) \land (\bigvee_{i=1}^{|S_c|} RIA_i)$ *represent the tuples which are dominated by* t_1.

Proof As mentioned above, the value b_i is determined as the minimum integer value satisfying $ITV_i[b_i] > t_1.A_i$. Therefore, the bit 1s of $\neg MCR_{i,b_i}$ represent the tuples whose A_i values are greater than $t_1.A_i$. Since we treat the incomplete attribute values as positive infinity, $\bigwedge_{i=1}^{|S_c|} \neg MCR_{i,b_i}$ represents the tuples whose values of $A_1, \dots, A_{|S_c|}$ are all greater than those of t_1. Given t_2 among these tuples, if at least one of $A_1, \dots, A_{|S_c|}$ of t_2 is complete attribute, t_2 is dominated by t_1 according to the dominance definition over incomplete data. If all of $A_1, \dots, A_{|S_c|}$ of t_2 are incomplete, t_1 and t_2 are not comparable from the perspective of dominance relationship.

The bit 1s of $\bigvee_{i=1}^{|S_c|} RIA_i$ mean that at least one of $A_1, \ldots, A_{|S_c|}$ is complete, and the bit 0s of $\bigvee_{i=1}^{|S_c|} RIA_i$ indicate that all of $A_1, \ldots, A_{|S_c|}$ are incomplete. Consequently, the bit 1s of $DBV_{t_1} = (\bigwedge_{i=1}^{|S_c|} \neg MCR_{i,b_i}) \wedge (\bigvee_{i=1}^{|S_c|} RIA_i)$ represent the tuples which are dominated by t_1. Q.E.D.

6.2.3 The Extraction of the Pruning Tuples

In order to skip the unnecessary tuples of T in stage 1, we first extract some pruning tuples for the following execution of TSI. The number of pruning tuples should not be large and they should have relatively strong dominance capability. Since the dimensionality of T can be high, we do not extract the pruning tuples with respect to the combination of different attributes, but to the values of single attribute and the number of complete attributes for each tuple. It is known that the cardinality of skyline results grows exponentially with the size of skyline criteria [4] and on incomplete data, dominance relationship between two tuples is performed over their common complete attributes. Intuitively, for a tuple, if it has a small number of complete attributes and one of its complete attributes is very small, it tends to have a relatively strong dominance capability.

The pruning tuples can be extracted from M sorted column files $SC_1, SC_2, \cdots, SC_M$. The schema of $SC_i (1 \leq i \leq M)$ is (PI_T, NUM_c, A_i), where NUM_c is the number of the complete attributes for each tuple. The tuples of $SC_i (1 \leq i \leq M)$ are sorted on NUM_c and A_i, i.e., they are first arranged in the ascending order of NUM_c, then all tuples with the same NUM_c are arranged in the ascending order of A_i.

For each sorted column file SC_i, we retrieve its tuples sequentially. Let sc be the current retrieved tuple, if $sc.A_i$ is within the first $f\%$ proportion among all A_i values, the PI_T value of sc is maintained in memory, and otherwise, the next tuple is retrieved. The process continues until the number of PI_T values maintained in memory reaches n_{pt} or it reaches to the end of file. Then, the corresponding tuples of T are extracted and kept in a separate pruning tuple file PT_i. In this paper, f is set to 5 and n_{pt} is set 1000; the pruning effect with such parameter setting is satisfactory in the performance evaluation.

Example 4 Figure 8 illustrates the extracting of pruning tuples in the running example. $SC_i (1 \leq i \leq 3)$ is arranged first in the ascending order of NUM_c, and the tuples with the same value of NUM_c are sorted in ascending of A_i. In the running example, $f = 12.5 (16 \times 12.5\% = 2)$ and $n_{pt} = 1$, one pruning tuple will be retrieved for SC_i. For SC_1, $SC_1(1, \ldots, 11)$ cannot be used to generate pruning tuples since their attribute values are not within the first two smallest values of A_i. Then, $SC_1(12)$ is selected to obtain the pruning tuple $T(SC_1(12).PI_T)$ since it is the first tuple in SC_1 whose A_1 value is among the first two smallest values of A_1. Other pruning tuples ($T(14)$ and $T(6)$) are obtained similarly.

	PI	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SC_1	PI_T	3	4	15	13	5	11	7	2	10	14	1	12	8	6	9	16
	NUM_c	1	1	2	2	2	2	2	2	2	2	2	3	3	3	3	3
	A_1	48	∞	39	47	56	65	67	71	73	86	∞	20	26	56	78	90
SC_2	PI_T	4	3	14	10	1	7	2	11	5	13	15	6	8	9	12	16
	NUM_c	1	1	2	2	2	2	2	2	2	2	2	3	3	3	3	3
	A_2	82	∞	6	50	91	94	∞	∞	∞	∞	∞	3	20	22	71	79
SC_3	PI_T	3	4	15	1	2	13	5	11	14	7	10	6	16	9	8	12
	NUM_c	1	1	2	2	2	2	2	2	2	2	2	3	3	3	3	3
	A_3	∞	∞	21	46	51	78	80	87	∞	∞	∞	0	8	28	38	54

$f = 12.5, n_{pt} = 1$

current tuples in SC_1	*operation*
$SC_1(1, \ldots, 11)$	$\forall 1 \leq a \leq 11, SC_1(a).A_1 > ITV_1(1) = 26$
$SC_1(12) = (12, 3, 20)$	$SC_1(12).A_1 = 20 \leq ITV_1[1] = 26$

current tuples in SC_2	*operation*
$SC_2(1, \ldots, 2)$	$\forall 1 \leq a \leq 2, SC_2(a).A_2 > ITV_2(1) = 6$
$SC_2(3) = (14, 2, 6)$	$SC_2(3).A_2 = 6 \leq ITV_2[1] = 6$

current tuples in SC_3	*operation*
$SC_3(1, \ldots, 11)$	$\forall 1 \leq a \leq 11, SC_3(a).A_3 > ITV_3(1) = 8$
$SC_3(12) = (6, 3, 0)$	$SC_3(12).A_3 = 0 \leq ITV_3[1] = 8$

PT_1: $T(12)(20, 71, 54)$ PT_2: $T(14)(86, 6, -)$ PT_3: $T(6)(56, 3, 0)$

Fig. 8 Illustration of extracting pruning tuples in the running example

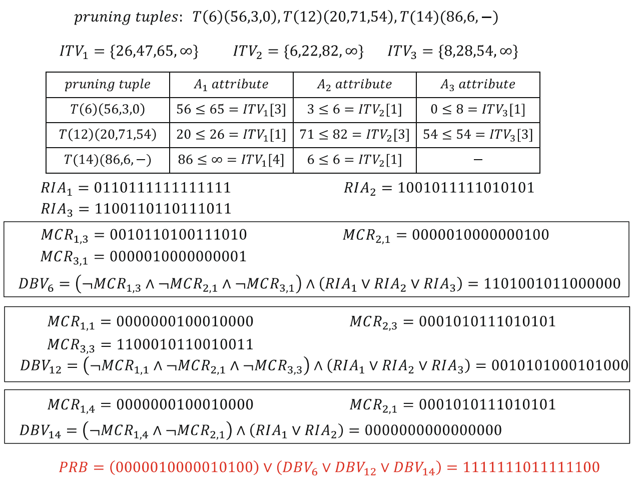
6.2.4 The Execution of Pruning Operation

By the pre-constructed structures described above, TSI can utilize pruning operation to reduce the execution cost in stage 1. In order to execute the pruning operation, TSI maintains a n-bit pruning bit-vector PRB in memory, which is filled with bit 0 initially.

Algorithm 2 TSI_Pruning(T, S_{cnd})

```
Input: $T$ is an incomplete table, $S_{cnd}$ a set maintaining the candidate tuples
Output: $S_{cnd}$ a set maintaining the skyline tuples over $T$
1: $MH$ is a min-heap to keep $mpruningtuples$ with the highest dominance capability.
2: initialize $S_{cnd} \leftarrow \emptyset$, $MH \leftarrow \emptyset$;
3: // Stage 1 find the candidate tuples
4: extract the involved pruning tuples $PT_1$, $PT_2$, ..., $PT_m$ for each skyline criteria of $T$,
   and put $PT_1$, $PT_2$, ..., $PT_m$ in to $MH$;
5: while $MH$ has more pruning tuples do
6:     retrieve the next tuple $pt$ of $MH$;
7:     $S_c$ is the complete attributes of $pt$, $S_c = \{A_1, \ldots, A_{|S_c|}\}\}$;
8:     if PRB(pt)=1 then
9:         pt can be skipped;
10:    else
11:        for ($i = 1$; $i \leq |S_c|$; $i++$) do
12:            compute the first value $ITV_i[b_i]$ of $ITV_i$, $ITV_i[b_i] \leftarrow SL_i(2^{b_i}).A_i$;
13:        end for
14:        the $(pt.PI_T)^{th}$ bit of $PRB$ to be 1;
15:        if $S_{cnd} = \emptyset$ then
16:            $S_{cnd} \leftarrow S_{cnd} \cup t$;
17:        else
18:            while $S_{cnd}$ has more tuples do
19:                retrieve the next tuple $p$ of $S_{cnd}$;
20:                if $p$ is dominated by $t$ then
21:                    remove $p$ from $S_{cnd}$;
22:                end if
23:            end while
24:            if $t$ is dominated by $p$ then
25:                discard $t$;
26:            else
27:                $S_{cnd} \leftarrow S_{cnd} \cup t$;
28:            end if
29:            $DBV_{pt} \leftarrow (\bigwedge_{i=1}^{|S_c|} \neg MCR_{i,b_i}) \wedge (\bigvee_{i=1}^{|S_c|} RIA_i)$;
30:            $PRB = PRB \vee (\bigvee_{b=1}^{m} DBV_{pt})$;
31:        end if
32:    end if
33: end while
34: // Stage 2 discard the candidates which are dominated by some tuples
35: while $T$ has more tuples do
36:     retrieve the next tuple $t$ of $T$;
37:     while $S_{cnd}$ has more tuples do
38:         retrieve the next tuple $can$ of $S_{cnd}$;
39:         if $can$ is dominated by $t$ then
40:             remove $can$ from $S_{cnd}$;
41:         end if
42:     end while
43: end while
44: return $S_{cnd}$;
```

Algorithm 2 is the pseudo-code of the execution of pruning operation. At the beginning of the stage 1, TSI determines the involved pruning tuple files $PT_1, PT_2, \ldots, PT_m$ according to the current skyline criteria and retrieves pruning tuples from them. In the process of retrieving $PT_1, PT_2, \ldots, PT_m$, TSI maintains a min-heap MH in memory to keep *m pruning tuples with the highest dominance capability* (line 4). Given a pruning tuple pt, let S_c be its complete attributes. Likewise, assume that $S_c = \{A_1, \ldots, A_{|S_c|}\}\}$ (line 5-6). $\forall 1 \leq i \leq |S_c|$, we determine the first value $ITV_i[b_i]$ of ITV_i which is greater than $pt.A_i$, its dominance capability is computed to be $\prod_{i=1}^{|S_c|} \frac{2^{b_i}}{n}$ (line 11-13). For the retrieved pruning tuple pt, TSI sets the $(pt.PI_T)$th bit of PRB to be 1, since it is retrieved already (line 14). Besides, for each pruning tuple pt, TSI removes any candidates in S_{cnd} which are dominated by pt (line 18-23). If pt is not dominated by any candidate in S_{cnd}, TSI keeps it in S_{cnd} (line 26-27). $\forall pt_b \in MH (1 \leq b \leq m)$, TSI computes its corresponding bit-vector DBV_{pt_b} of dominance checking as in Sect. 6.2.2 (line 29). The final pruning bit-vector PRB is $PRB = PRB \vee (\bigvee_{b=1}^{m} DBV_{pt_b})$ (line 30).



pruning tuples: $T(6)(56,3,0), T(12)(20,71,54), T(14)(86,6,-)$

$ITV_1 = \{26,47,65,\infty\}$ $ITV_2 = \{6,22,82,\infty\}$ $ITV_3 = \{8,28,54,\infty\}$

pruning tuple	A_1 *attribute*	A_2 *attribute*	A_3 *attribute*
$T(6)(56,3,0)$	$56 \leq 65 = ITV_1[3]$	$3 \leq 6 = ITV_2[1]$	$0 \leq 8 = ITV_3[1]$
$T(12)(20,71,54)$	$20 \leq 26 = ITV_1[1]$	$71 \leq 82 = ITV_2[3]$	$54 \leq 54 = ITV_3[3]$
$T(14)(86,6,-)$	$86 \leq \infty = ITV_1[4]$	$6 \leq 6 = ITV_2[1]$	$-$

$RIA_1 = 0110111111111111$ $RIA_2 = 1001011111010101$
$RIA_3 = 1100110110111011$

$MCR_{1,3} = 0010110100111010$ $MCR_{2,1} = 0000010000000100$
$MCR_{3,1} = 0000010000000001$
$DBV_6 = (\neg MCR_{1,3} \wedge \neg MCR_{2,1} \wedge \neg MCR_{3,1}) \wedge (RIA_1 \vee RIA_2 \vee RIA_3) = 1101001011000000$

$MCR_{1,1} = 0000000100010000$ $MCR_{2,3} = 0001010111010101$
$MCR_{3,3} = 1100010110010011$
$DBV_{12} = (\neg MCR_{1,1} \wedge \neg MCR_{2,1} \wedge \neg MCR_{3,3}) \wedge (RIA_1 \vee RIA_2 \vee RIA_3) = 0010101000101000$

$MCR_{1,4} = 0000000100010000$ $MCR_{2,1} = 0001010111010101$
$DBV_{14} = (\neg MCR_{1,4} \wedge \neg MCR_{2,1}) \wedge (RIA_1 \vee RIA_2) = 0000000000000000$

$PRB = (0000010000010100) \vee (DBV_6 \vee DBV_{12} \vee DBV_{14}) = 1111111011111100$

Fig. 9 Illustration of constructing PRB in the running example

Example 5 The construction of PRB in the running example is illustrated in Fig. 9. For the pruning tuple $T(6)(56, 3, 0)$, TSI determines $MCR_{1,3}$, $MCR_{2,1}$, $MCR_{3,1}$ which correspond to the values of $T(6)$. The tuples dominated by $T(6)$ can be specified by a bit-vector $PRB_6 = 1101001011000000$. Similarly, we obtain PRB_{12} and PRB_{14}. Since $T(6)$, $T(12)$, $T(14)$ are the pruning tuples, after retrieving them, PRB is set to be 0000010000010100, i.e., the 6th bit, the 12th and the 14th bit are 1. The final pruning bit-vector $PRB = PRB \vee (DBV_6 \vee DBV_{12} \vee DBV_{14}) = 1111111011111100$.

In stage 1, $\forall 1 \leq a \leq n$, if $PRB(a) = 1$, $T(a)$ can be skipped; otherwise, TSI needs to retrieve $T(a)$. The rest of the execution in stage 1 is the same as that in Sect. 6.1.

Example 6 In the running example, TSI only needs to retrieve three tuples ($T(8)$, $T(15)$, $T(16)$) in stage 1 by use of PRB. This reduces the I/O cost and computation cost significantly.

7 Performance Evaluation

7.1 Experimental Settings

To evaluate the performance of TSI, we implement it in Java with jdk-8u20-windows-x64. The experiments are executed on LENOVO ThinkCentre M8400 (Intel (R) Core(TM) i7 CPU @ 3.40GHz (8 CPUs) + 32G memory + 3TB HDD + 64 bit windows 7). In the experiments, we implement TSI, BA, SOBA [10] and SIDS [1]. With the experimental setting below, the execution time of SOBA and SIDS is so long that we do not report its experimental results with the settings below, but evaluate it in Sect. 7.8 separately. For BA, the size S of the allocated memory is 4GB. We do not use a larger size for BA because, with the assistance of the bit-vector B_{ret} as mentioned in Sect. 5, the larger value of S makes more tuples of T loaded in memory at a time and reduces the number of iteration, but it also reduces the proportion of retrieval which can use the optimization of skipping operation.

Table 2 Parameter Settings

Parameter	Used values
Tuple number(10^6) (syn)	5, 10, 50, 100, 500
Skyline criteria size (syn)	10, 15, 20, 25
Incomplete ratio (syn)	0.3, 0.4, 0.5, 0.6, 0.7
Correlation coefficient (syn)	-0.8, -0.4, 0, 0.4, 0.8
Incomplete ratio (real)	0.3, 0.4, 0.5, 0.6, 0.7

In the experiments, we evaluate the performance of TSI in terms of several aspects: tuple number (n), used attribute number (m), incomplete ratio (p), correlation coefficient (c). The experiments are executed on three data sets: two synthetic data sets (independent distribution and correlated distribution) and a real data set. The used parameter settings are listed in Table 2. For correlated distribution, the first two attributes have the specified correlation coefficient, while the left attributes follow the independent distribution. In order to generate two sequences of random numbers with correlation coefficient c, we first generate two sequences of uncorrelated distributed random number X_1 and X_2, then a new sequence $Y_1 = c \times X_1 + \sqrt{1 - c^2} \times X_2$ is generated, and we get two sequences X_1 and Y_1 with the given correlation coefficient c. When generating synthetic data, we fix the number of M to be 60 and generate data with all complete attributes. Then, according to used skyline criteria, we select one attribute first, this attribute is complete. Other $(m - 1)$ attributes in skyline criteria have a probability p of being incomplete independently. The real data used are HIGGS Data Set from UCI Machine Learning Repository[1], it is provided to classification problem including 11000000 instances. The main reasons for using HIGGS are that 1) HIGGS is one of the largest databases to our knowledge, accordingly, we have better access to compare the performance of above algorithms. 2) and it is an open dataset that we can find and obtain expediently. On real data, we evaluate the performance of TSI with varying values of p.

The required structures are pre-constructed before the experiments. Under the default setting of the experiments, i.e., $M = 60$, $n = 50 \times 10^6$, and $p = 0.3$, it takes 6840.573 seconds to pre-construct the required data structures.

[1] https://archive.ics.uci.edu/ml/datasets/HIGGS#

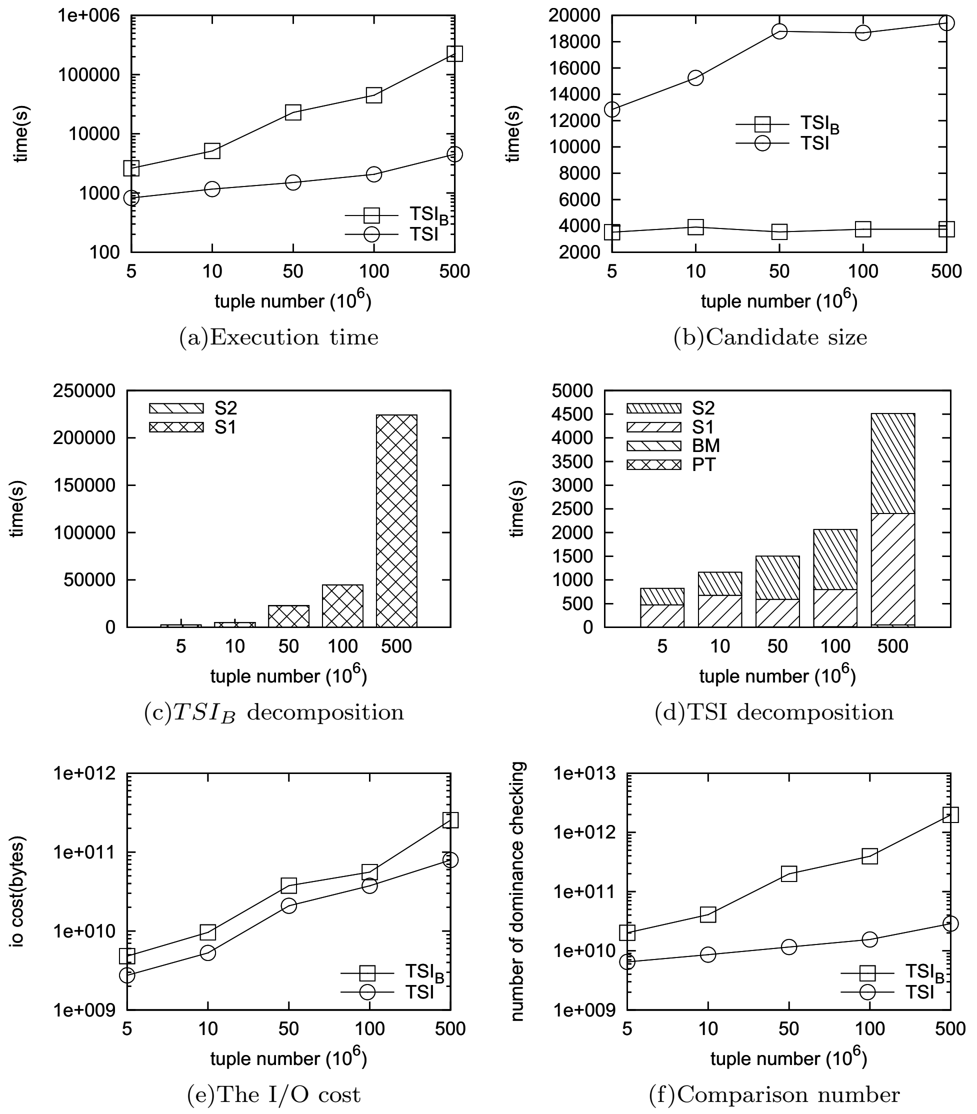


Fig. 10 Comparison between TSI_B and TSI

7.2 The Comparison of TSI with and Without Pruning

The performance of TSI_B and TSI is compared in different aspects, where TSI_B is the TSI algorithm without pruning operation. As depicted in Fig. 10a, TSI runs 18.84 times faster than TSI_B and the speedup ratio increases with a greater value of n. This significant advantage is due to the effective pruning operation. The numbers of the candidates after stage 1 are illustrated in Fig. 10b. TSI maintains more candidates than TSI_B after stage 1. This is because the pruning operation skips most of the tuples in stage 1, and therefore, many candidates which should be removed by some tuples are left. But the pruning operation reduces the cost in stage 1 significantly. Figure 10c reports the time decomposition of TSI_B. Obviously, the execution time of stage 1 dominates its overall time. We even cannot see the time in stage 2 due to its rather small proportion. Figure 10d gives the time decomposition of TSI, which consists of four parts: the time to retrieve pruning tuples, the time to load the required bit-vectors, the time in stage 1, and the time in stage 2. The time in stage 2 of TSI is longer than that of TSI_B due to the greater number of candidates left. However, the time reduction in stage 1 of TSI is much significant compared with TSI_B and TSI runs one order of magnitude faster than TSI_B averagely. As shown in Fig. 10(e and f), the pruning operation makes TSI incur less I/O cost and perform fewer number of dominance checking.

7.3 Experiment 1: the Effect of Tuple Number

Given $m = 20$, $M = 60$, $p = 0.3$ and $c = 0$, experiment 1 evaluates the performance of TSI on varying tuple numbers. As shown in Fig. 11a, TSI runs 60.42 times faster than BA averagely. The speedup ratio of TSI over BA increases with a greater value of n, from 8.31 at $n = 5 \times 10^6$ to 166.58 at

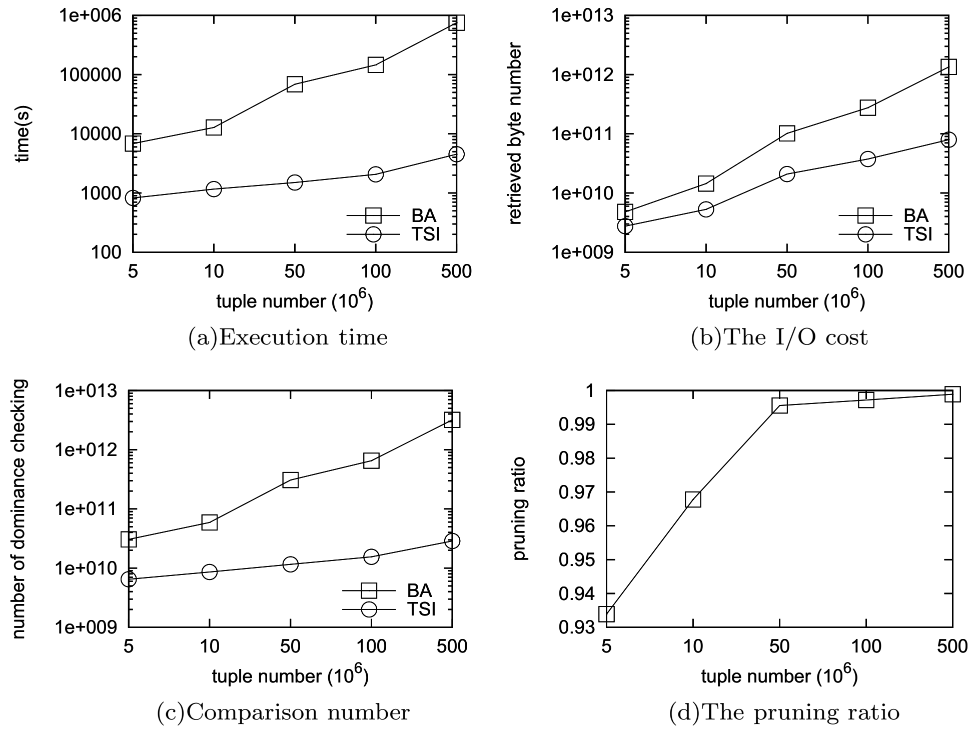


Fig. 11 Effect of tuple number

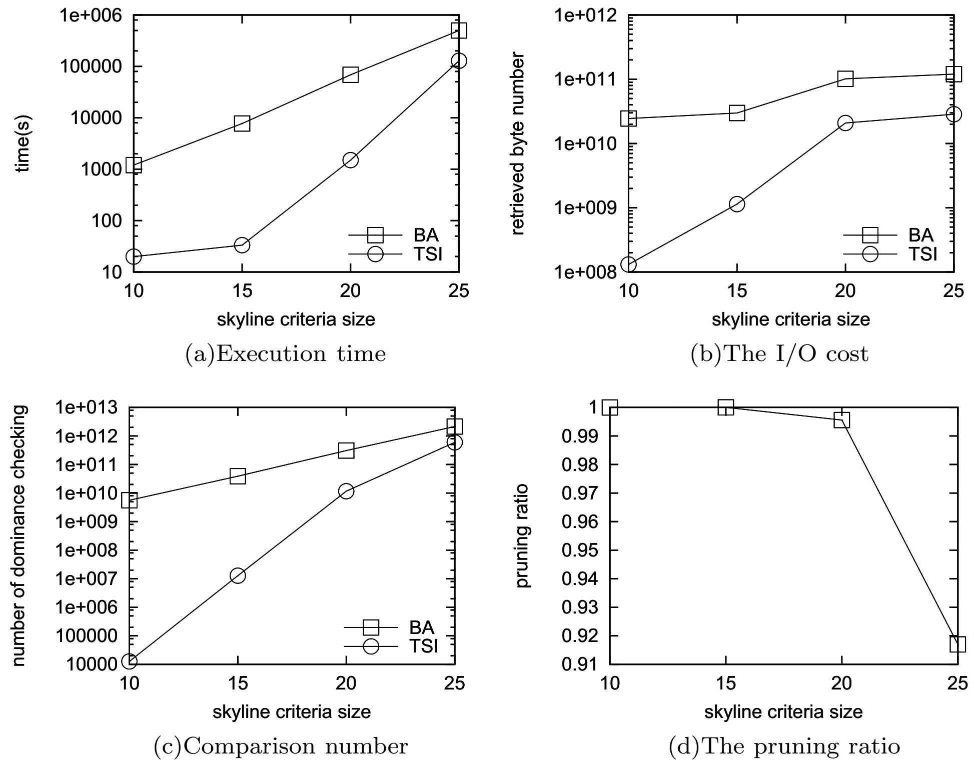
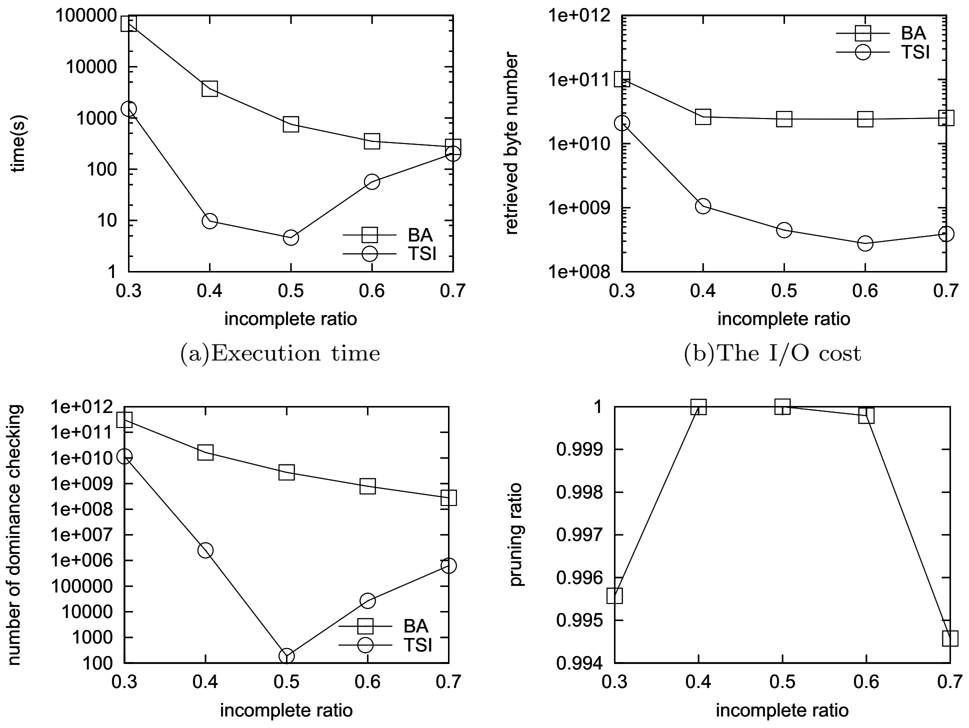


Fig. 12 Effect of skyline criteria size

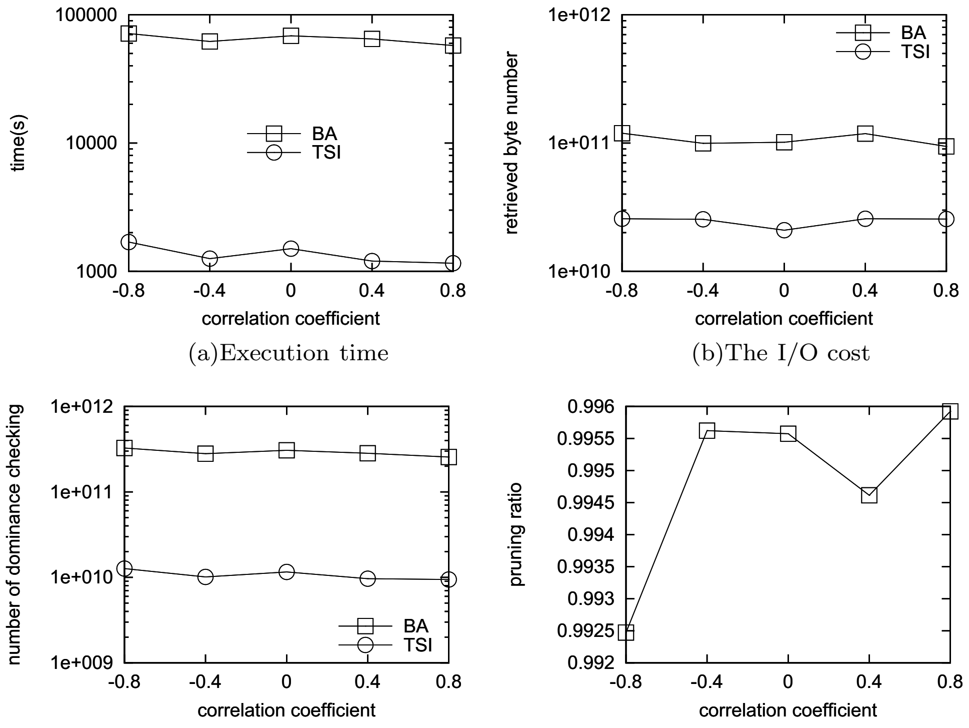
$n = 500 \times 10^6$. Figure 11b depicts that TSI incurs 6.73 times less I/O cost than BA. And as illustrated in Fig. 11c, TSI performs 38.17 times fewer number of dominance checking than BA. The performance advantage of TSI over BA is widened with the greater value of n. At $n = 5 \times 10^6$, BA can load all T into memory and perform another table scan on

Fig. 13 Effect of incomplete ratio



(a)Execution time (b)The I/O cost (c)Comparison number (d)The pruning ratio

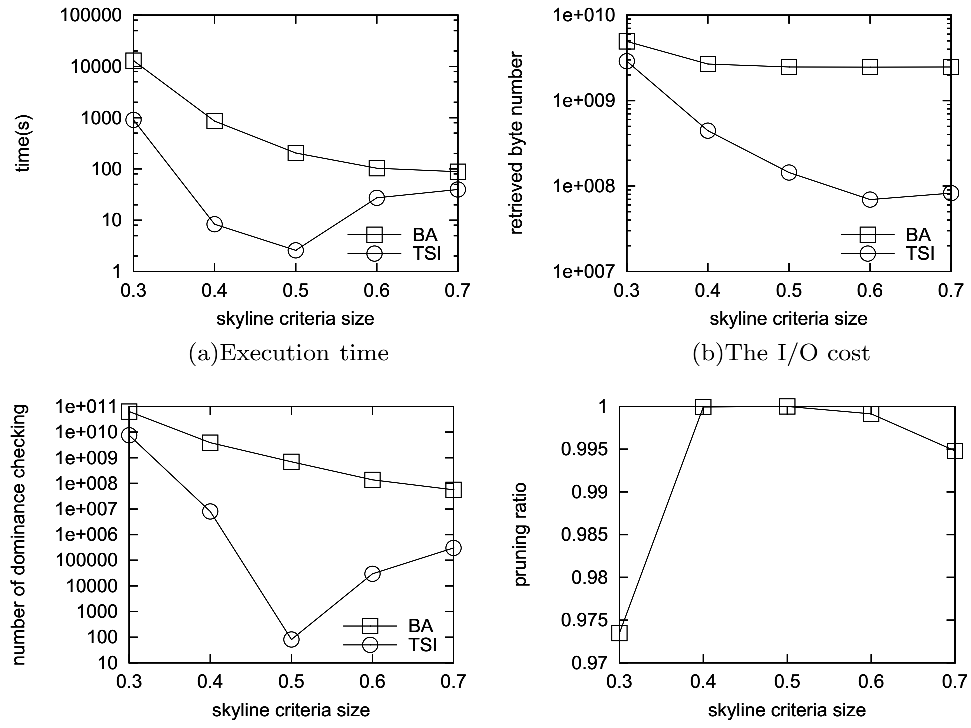
Fig. 14 Effect of correlation coefficient



(a)Execution time (b)The I/O cost (c)Comparison number (d)The pruning ratio

T to compute incomplete skyline results. At $n = 500 \times 10^6$, BA needs to execute 56 iterations, each loading a part of *T* and then followed by a table scan on *T* to remove the dominated tuples. On the contrary, TSI shows a slower growing trend on tuple number due to its execution process and pruning operation. As illustrated in Fig. 11d, the pruning operation of TSI can skip vast majority of tuples in stage 1. The pruning ratio in the experiments is computed by the

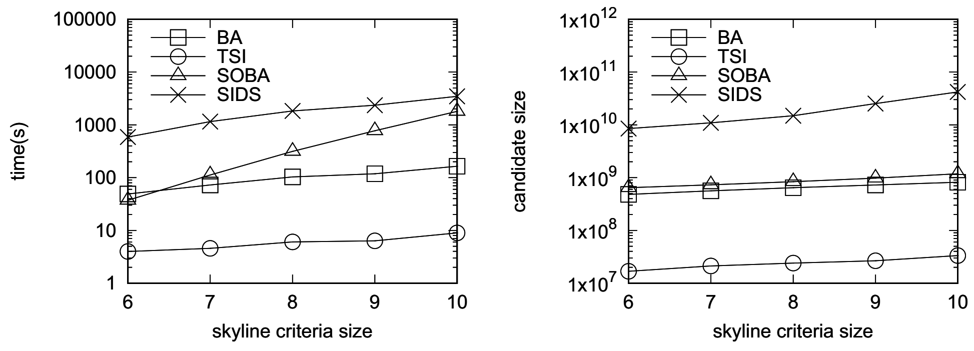
Fig. 15 Effect of real data



(a)Execution time (b)The I/O cost

(c)Comparison number (d)The pruning ratio

Fig. 16 Comparison with BA, SOBA, and SIDS



(a)Execution time (b)The I/O cost

formula $\frac{n_{skip}}{n}$, where n_{skip} is the number of tuples skipped in stage 1.

7.4 Experiment 2: the Effect of Skyline Criteria Size

Given $M = 60$, $n = 50 \times 10^6$, $p = 0.3$ and $c = 0$, experiment 2 evaluates the performance of TSI on varying skyline criteria sizes. As illustrated in Fig. 12a, with a greater value of m, the execution times of BA and TSI both increase significantly; TSI still runs 85.79 times faster than BA averagely. For BA, its I/O cost depends on two parts. For one thing, BA needs to retrieve T once to load it into memory. For another, BA performs a sequential scan on T in each iteration to discard the candidates in memory which are dominated by some tuples. For the first part, BA may not retrieve all tuples into memory since the current tuples may be dominated by the previous iterations. For the second part, if the current candidates all are discarded, BA does not have to continue the sequential scan but just performs the next iteration directly. When the value of m increases, given other parameters are fixed, the probability that a tuple is dominated by other tuple becomes lower. Therefore, the I/O cost increases on both parts. This is reported in Fig. 12b. For TSI, its I/O cost also consists of two parts. In stage 1, TSI performs a selective scan on T to obtain the candidates of incomplete skyline results. In stage 2, TSI does another sequential scan on T to compute the results, in which if all candidates are removed, TSI can terminate directly. As the value of m increases, the pruning effect in TSI becomes worse in stage 1, which also is verified in Fig. 12d, and TSI has to retrieve more tuples before it terminate in stage 2. This makes a higher I/O cost for TSI with a greater value of m, as illustrated in Fig. 12b. With the similar explanation, as shown in Fig. 12c, the numbers of dominance checking for both algorithms increase with a greater value of m.

7.5 Experiment 3: the Effect of Incomplete Ratio

Given $m = 20$, $M = 60$, $n = 50 \times 10^6$ and $c = 0$, experiment 3 evaluates the performance of TSI on varying incomplete ratios. As the value of p increases, the execution time of BA decreases quickly, while the execution time of TSI first decreases and then increases gradually. For BA, the decline of execution time is easy to understand. With a greater value of p, the probability that any tuple is dominated by other tuples increases. This makes more in-memory candidates in each iteration dominated by some tuples in the sequential scan, and can reduce the I/O cost and dominance checking cost. As illustrated in Fig. 13c, with a greater value of p, the number of dominance checking in BA decreases constantly. And as shown in Fig. 13b, the I/O cost of BA first decreases significantly when p increases from 0.3 to 0.4, then remains unchanged basically ever since. When p increases from 0.3 to 0.4, the number of in-memory candidates is reduced during the sequential scan and in each iteration, BA terminates earlier. This makes less I/O cost for BA. When the value of p is greater than 0.4, the number of in-memory candidates is reduced also, but in each iteration, BA reaches an approximately equal scan depth before it terminates. For TSI, the effect of pruning operation depends on two factors. One is the probability that one tuple can be dominated by other tuples. The other is whether all common attributes of two tuples are incomplete. The two factors have different effects in different cases. With a greater value of p, the probability of a tuple dominated by some tuples increases, also the probability that the common attributes of two tuples are all incomplete. When p increases from 0.3 to 0.5, the first factor has a greater impact, and ever since, the second factor plays a larger role. This explains the trend of the execution time of TSI. Similarly, this can explain the variation trend of TSI in I/O cost (Fig. 13b), the number of dominance checking (Fig. 13c), and the pruning ratio (Fig. 13d).

7.6 Experiment 4: the Effect of Correlation Coefficient

Given $m = 20$, $M = 60$, $n = 50 \times 10^6$ and $p = 0.3$, experiment 4 evaluates the performance of TSI on varying correlation coefficients. As illustrated in Fig. 14a, TSI runs 47.72 times faster than BA. The correlation coefficients considered range from -0.8 to 0.8. A negative correlation means that there is an inverse relationship between two variables, when one variable decreases, the other increases. And a positive correlation means that variables tend to move in the same direction. Therefore, the skyline computation on negatively correlated data usually is more expensive than that on positively correlation data. The variations in TSI and BA both show a downward trend in experiment 4. Here, the trend is not significant because the incomplete attributes in the data set reduce the impact of correlation. The I/O cost and number of dominance checking are depicted in Fig. 14(b and c), respectively, and they have the similar variation trends. The effect of pruning operation of TSI is illustrated in Fig. 14d. Due to the impact of incomplete attributes, the pruning ratio shows considerable change, but it still shows upward trend overall.

7.7 Experiment 5: Real Data

The real data, HIGGS Data Set, are obtained from UCI Machine Learning Repository. It contains 11,000,000 tuples with 28 attributes. We select the first 20 attributes as skyline criteria and evaluate the performance of TSI with varying incomplete ratios. Before the experiment is executed, one attribute first is chosen to be complete and other $(m - 1)$ attributes in skyline criteria have a probability p of being incomplete independently. As depicted in Fig. 15a, TSI runs 40.46 times faster than BA. The variation trends of execution times of BA and TSI are very close to those in Sect. 7.5 and can be explained similarly. The I/O cost and the number of dominance checking are depicted in Fig. 15(b and c), respectively. The pruning ratio in TSI is illustrated in Fig. 15d. The variation in these figures can be explained similarly as in Sect. 7.5.

7.8 Experiment 6: the Comparison with SOBA and SIDS

In this part, we evaluate the performance of TSI against BA, SOBA and SIDS on a relatively small data set with relatively small skyline criteria size. Given $n = 10 \times 10^6$, $p = 0.3$ and $c = 0$, in order to acquire a better performance for SOBA and SIDS, we set the value of m to be from 6 to 10, and the value of M equal to that of m. This can reduce the length of each tuple and also lower the cost of bucket partitioning for SOBA and SIDS.

As illustrated in Fig. 16a, SIDS is the slowest among the four algorithms while TSI is the faster in various skyline criteria size, and the execution time of SOBA increases significantly with the number of m. When $m = 10$, SOBA runs 10.96 times slower than BA, the baseline algorithm in this paper, and runs 200.91 times slower than TSI. As for SIDS, it runs 21.11 times slower than BA and runs 386.84 times slower than TSI. On disk resident data, SOBA and SIDS cannot process incomplete skyline efficiently. The bucket partitioning of SOBA involves two passes of table scan, not to mention the maintenance cost of the large number of partitions in the disk if the number of m is not small. Then, the computation of local skyline involves another pass of tuple retrieval. On the relatively large value of m, the number of local skyline is great also. As depicted in Fig. 16b, the local skyline makes up 11.7% of the total tuples at $m = 10$. The I/O cost of SIDS is much larger than others, SIDS and BA are much close in I/O cost. The growth trend of the execution time of SIDS is fast with respect to skyline criteria size. The performance of TSI is efficient not only for the in-memory data set with small size of skyline criteria, but also for the disk-resident data with not small size of skyline criteria.

8 Conclusion

This paper considers the problem of incomplete skyline computation on massive data. It is analyzed that the existing algorithms cannot process the problem efficiently. A table-scan-based algorithm TSI is devised in this paper to deal with the problem efficiently. Its execution consists of two stages. In stage 1, TSI maintains the candidates by a sequential scan. And in stage 2, TSI performs another sequential scan to refine the candidate and acquire the final results. In order to reduce the cost in stage 1, which dominates the overall cost of TSI, a pruning operation is utilized to skip the unnecessary tuples in stage 1. The experimental results show that TSI outperforms the existing algorithms significantly.

References

1. Bharuka R, Sreenivasa Kumar P (2013) Finding skylines for incomplete data. In: Proceedings of the 24th australasian database conference - Vol 137, pp 109–117
2. Börzsönyi S, Kossmann D, Stocker K (2001) The skyline operator. In: Proceedings of the 17th international conference on data engineering, pp 421–430
3. Chomicki J, Godfrey P, Gryz J, Liang D (2003) Skyline with presorting. In: Proceedings of the 19th international conference on data engineering, pp 717–719
4. Godfrey P (2004) Skyline cardinality for relational processing. In: foundations of information and knowledge systems, Third International Symposium, FoIKS 2004:78–97
5. Godfrey Parke, Shipley Ryan, Gryz Jarek (2007) Algorithms and analyses for maximal vector computation. VLDB J 16(1):5–28
6. Xixian H, Jianzhong L, Donghua Y, Jinbao W (2013) Efficient skyline computation on big data. IEEE Trans Knowl Data Eng 25(11):2521–2535
7. Khalefa ME, Mokbel MF, Levandoski JJ (2008) Skyline query processing for incomplete data. In: Proceedings of the 24th international conference on data engineering, pp 556–565
8. Kossmann D, Ramsak F, Rost S (2002) Shooting stars in the sky: an online algorithm for skyline queries. In: Proceedings of the 28th international conference on very large data bases, pp 275–286
9. Lee Jongwuk, Hwang Seung-Won (January 2014) Scalable skyline computation using a balanced pivot selection technique. Inf Syst 39:1–21
10. Lee Jongwuk, Im Hyeonseung, You Gae-won (2016) Optimizing skyline queries over incomplete data. Inf Sci 361–362:14–28
11. Lee Ken C, Lee Wang-Chien, Zheng Baihua, Li Huajing, Tian Yuan (2010) Z-sky: an efficient skyline query processing framework based on z-order. VLDB J 19(3):333–362
12. Luo Cheng, Jiang Zhewei, Hou Wen-Chi, He Shan, Zhu Qiang (2012) A sampling approach for skyline query cardinality estimation. Knowl Inf Syst 32(2):281–301
13. Miao X, Yunjun G, Su G, Wanqi L (2018) Incomplete data management: a survey. Front Comput Sci 12(1):4–25
14. Papadias Dimitris, Tao Yufei, Greg Fu, Seeger Bernhard (2005) Progressive skyline computation in database systems. ACM Trans Database Syst 30(1):41–82
15. Sheng C, Tao Y(2011) On finding skylines in external memory. In: Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems, pp 107–116
16. Tan K-L, Eng P-K, Ooi BC (2001) Efficient progressive skyline computation. In: Proceedings of the 27th international conference on very large data bases, pp 301–310
17. Tao Yufei, Xiao Xiaokui, Pei Jian (2007) Efficient skyline and top-k retrieval in subspaces. IEEE Trans Knowl Data Eng 19(8):1072–1088
18. Zhang K, Gao H, Han X, Cai Z, Li J (2017) Probabilistic skyline on incomplete data. In: Proceedings of the 2017 ACM on conference on information and knowledge management, pp 427–436
19. Zhang Kaiqi, Gao Hong, Han Xixian, Cai Zhipeng, Li Jianzhong (2020) Modeling and computing probabilistic skyline on incomplete data. IEEE Trans Knowl Data Eng 32(7):1405–1418
20. Shiming Z, Nikos M, Cheung DW (2009) Scalable skyline computation using object-based space partitioning. In: Proceedings of the 2009 ACM SIGMOD international conference on management of data, pp 483–494
21. Zhenjie Z, Hua L, Beng Chin O, Tung AK (2010) Understanding the meaning of a shifted sky: a general framework on extending skyline query. The VLDB J 19(2):181–201
22. Zhenjie Z, Yin Y, Ruichu C, Dimitris P, Anthony KHT (2009) Kernel-based skyline cardinality estimation. In: Proceedings of the ACM SIGMOD international conference on management of data, pp 509–522