IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 12, 2021, accepted April 25, 2021, date of publication May 11, 2021, date of current version May 24, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3079324

CrowdSJ: Skyline-Join Query Processing of Incomplete Datasets With Crowdsourcing

LINLIN DING[1,2], XIAO ZHANG[1,2], HANLIN ZHANG[1,2], LIANG LIU[1], AND BAOYAN SONG[1,2]
[1]School of Information, Liaoning University, Shenyang 110036, China
[2]Mine Dynamic Disaster Monitoring and Warning Big Data Engineering Research Center, Shenyang 110036, China

Corresponding author: Baoyan Song (bysong@lnu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 62072220 and Grant 61502215, in part by the Science Research Normal Fund of Liaoning Province Education Department under Grant LJC201913, and in part by the China Postdoctoral Science Foundation Funded Project under Grant 2020M672134.

ABSTRACT Skyline query is very useful in decision-making systems, WSN and so on. As a variation of skyline query, skyline-join query can return the results from multiple datasets. However, incomplete datasets are a frequent phenomenon due to the widespread use of automated information extraction and aggregation. Existing methods for dealing with incomplete data, such as probability, data padding can solve the problem, but cannot effectively reflect the real situation and are lack of integrality. Therefore, in this paper, in order to reflect the situation more accuracy and more user-centric, we research the problem of skyline-join query over incomplete datasets with crowdsourcing, named CrowdSJ. The crowdsourcing-based skyline-join query processing problem over incomplete datasets is divided into two situations. One is the skyline-join query only involves the unknown crowdsourcing attribute and the join attribute, named Partial Skyline-Join with Crowdsourcing (PSJCrowd). The other one is the skyline-join query involves all the attributes, named All Skyline-Join with Crowdsourcing (ASJCrowd). For PSJCrowd, first, we filter the known dataset. Then, we present the level-preference-tree-index, and propose the partial skyline-join with crowdsourcing algorithm. For ASJCrowd, first, we filter the known dataset too. Second, we build a level-preference-tree-index based on the known attributes of the incomplete dataset. Third, we propose the skyline-join with crowdsourcing on single dataset algorithm, CrowdSJ-single, to filter the dataset containing unknown attributes. Then, we build a global level-preference-tree-index based on the known attributes of the incomplete dataset and the complete dataset. We propose the skyline-join with crowdsourcing on multiple datasets algorithm, CrowdSJ-multiple. We filter the linked tuples based on the global level-preference-tree-index and the results of each round of crowdsourcing. Numerous experiments on synthetic and real datasets demonstrate that our algorithms are efficient and effective.

INDEX TERMS Skyline-join query, incomplete data, crowdsourcing, index structure.

I. INTRODUCTION

Skyline query is very useful in decision-making systems [1], [2], Wireless Sensor Networks(WSN) [3]–[5], Navigation [6], Demographic [7], geographic services [8], and so on. Given a set of attributes of interest, a skyline query retrieves the tuples which cannot be dominated by others in any of the attributes. For example, to find a suitable hotel, a user may propose a query 'return the hotels which are both cheap and close to the beach'.

Existing research tends to assume that the skyline query is issued to a single dataset. That is, all the required attributes are from the same dataset. However, this assumption is no longer valid for the web environment, where data from multiple sources are required for query processing. In this situation, a variation of skyline query that is skyline-join query has been proposed. Skyline-join query can return the results from multiple datasets. The issue of skyline-join query has been extensively studied, and the representative works include [9], [20]–[23]. Figure 1 illustrates an example of computing the skyline-join over two datasets. Suppose that the database contains the relations Hotels and Restaurants that store information about a specific city. A tourist may be interested in discovering the best combinations of hotels and restaurants in the same 'Loc', by minimizing the hotel's 'H-Price' and 'Rating' (Supposing), and by minimizing

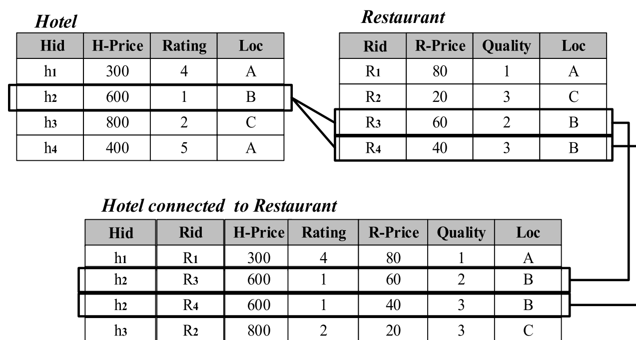
The associate editor coordinating the review of this manuscript and approving it for publication was Bilal Alatas.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

the restaurant's 'Quality' (Supposing) and its 'R-Price'. A sample SQL query that retrieves this information is as the following:

SELECT H.HID, R.RID, H.H-Price, R.R-Price,
H.Loc, H.Rating, R.Quality
FROM *Hotel* H, *Restaurant* R
WHERE H.Loc= R.Loc
SKYLINE OF H.Price MIN, H.Rating MIN,
R.R-Price MIN, R.Quality MIN

The result dataset of the skyline-join query is depicted in Figure 1. The use of the skyline operator allows the retrieval of the most '*important*' join tuples according to the specified attributes.



Hotel

Hid	H-Price	Rating	Loc
h_1	300	4	A
h_2	600	1	B
h_3	800	2	C
h_4	400	5	A

Restaurant

Rid	R-Price	Quality	Loc
R_1	80	1	A
R_2	20	3	C
R_3	60	2	B
R_4	40	3	B

Hotel connected to Restaurant

Hid	Rid	H-Price	Rating	R-Price	Quality	Loc
h_1	R_1	300	4	80	1	A
h_2	R_3	600	1	60	2	B
h_2	R_4	600	1	40	3	B
h_3	R_2	800	2	20	3	C

FIGURE 1. Example of skyline-join.

Normally, skyline-join query consists of two phases, joining the relations containing the required attributes and making a skyline query in the relation after the join. In skyline-join query in Figure 1, two datasets are first linked through the unique link attribute 'Loc', and then perform skyline query in the dataset after join. Tuple $\{h_4, R_1\}$ is dominated by $\{h_1, R_1\}$, and the remaining tuples after the join cannot be dominated by other tuples, so they are all the results of skyline-join query. It illustrates that skyline-join can return skyline results from multiple datasets. However, these results involve many attributes which are difficult for users to choose. At this time, if the overall evaluation of the restaurant is obtained through the previous users of this restaurant, named *cost-effective*, it can reflect the actual situation more accurately and be more user-centric.

However, the *cost-effective* of one restaurant is often unknown. Moreover, incomplete datasets are a frequent phenomenon due to the widespread use of automated information extraction and aggregation. The existing skyline-join query connects all relevant datasets and then applies the existing skyline algorithms based on complete dataset. It is difficult for the dataset with incomplete or unknown data attributes. Although, in certain situation, we can infer some information from the known attributes, this kind of inference often differs from the results expected. The more extreme situation is that a certain attribute in the dataset is completely missing, and there are cases where the attributes are partially missing in the actual application. Existing methods for dealing with incomplete data, such as probability [11], [16], data padding [12] can solve the problem, but cannot effectively reflect the real situation and are lack of integrality.

Recently, crowdsourcing [10], [35]–[39] has emerged as a new computing paradigm for human computation and has been widely used for bridging the gap between machine-based and human-based computation. Humans can realize considerably improved results compared to computers when performing intelligent tasks such as answering users' semantic search queries [29], [30], understanding topics in microblogs [31], [32], and images tagging for subjective topics [33], and so on. To solve the problem of skyline-join over incomplete datasets, we use crowdsourcing to effectively infer missing values on the supervisor attribute. Therefore, in this paper, we research the problem of skyline-join query over incomplete datasets with crowdsourcing, named CrowdSJ.

In this paper, crowdsourcing-based skyline-join query processing problem over incomplete datasets is divided into two situations. One situation is that for two datasets of skyline-join query, all the attributes of one dataset are known, and only one attribute of the other dataset is unknown, where the skyline-join query only involves the unknown crowdsourcing attribute and the join attribute, named Partial Skyline-Join with Crowdsourcing (PSJCrowd). For example, when we arrive somewhere, we want to find a hotel with better price and rating, and around it there are the restaurants with better *cost-effective*. The *cost-effective* of a restaurant is often unknown which needs to crowdsourcing. First, we filter the data tuples in the known dataset. Second, the tuples in the unknown dataset are filtered based on the join attribute and compared in pairs. Currently, since the skyline-join of the unknown dataset only involves one unknown attribute and a join attribute, the result of crowdsourcing is optimal in each group. Finally, a level-preference-tree-index is established based on the attributes of the known dataset [13], and then the tuples are filtered through the dominance of the known data attributes in the index and the crowdsourcing results of each round to obtain the global skyline-join results.

Another situation is that for two datasets of skyline-join query, all the attributes of one dataset are known, and only one attribute of the other dataset is unknown, where the skyline-join query involves all the attributes not only the unknown crowdsourcing attribute and the join attribute, named All Skyline-Join with Crowdsourcing (ASJCrowd). For example, when we arrive somewhere, we want to find a hotel with better price and rating, and around it there are the restaurants not only with better *cost-effective*, but also the other attributes, price, rating and so on. The *cost-effective* of a restaurant is often unknown which needs to crowdsourcing too. In this case, the first step is the same as the first case. The second step is to group meta-groups according to the different link attributes, and to index each group based on the known attributes in the unknown dataset. We filter the tuples based on the dominance of the index in the unknown dataset and the crowdsourcing results of each round in the unknown dataset. Finally, the two datasets are joined, and a

global level-preference-tree-index index is established based on the attributes of the known dataset and the known attributes in the unknown dataset. The skyline-join result is obtained by filtering the tuples in the unknown dataset based on the dominance of the index in all datasets and the crowdsourcing results of each round. The main contributions of this article are as follows:

- For Partial Skyline-Join with Crowdsourcing problem (PSJCrowd), we first introduce the level-preference-tree-index, and propose the partial skyline-join with crowdsourcing algorithm. The proposed algorithm filters the tuples based on the dominance between the tuples of a known dataset and the results of each round of crowdsourcing.
- For All Skyline-Join with Crowdsourcing problem (ASJCrowd), first we filter the known dataset. Then on the incomplete dataset, we build a level-preference-tree-index based on the known attributes of the incomplete dataset. The algorithm filters the dataset containing unknown attributes based on this index and each round of crowdsourcing results. Globally, we build a global level-preference-tree-index based on the known attributes of the incomplete dataset and the complete dataset. The algorithm we proposed filters the linked tuples based on the global level-preference-tree-index and the results of each round of crowdsourcing.
- Numerous experiments on synthetic and real datasets demonstrate that our algorithms are efficient and effective.

The organization of this paper is as follows. Section II describes the work related of skyline queries on incomplete datasets, crowdsourcing queries, and skyline-join queries. Section III researches the partial skyline-join with crowdsourcing problem (PSJCrowd). Section IV solves the all skyline-join with crowdsourcing problem (ASJCrowd). The performance of the algorithms is evaluated in Section V through experiments. Section VI summarizes the work of this paper.

II. RELATED WORK

A. SKYLINE-JOIN QUERY PROCESSING

Akrivi Vlachou *et al.* [23] proposed the novel Sort-First-Skyline-Join (SFSJ) algorithm that mixed the identification of skyline tuples with the computation of join, and efficiently computed the skyline set of a relational join. Nagendra M *et al.* Paper [28] proposed skyline-sensitive join (S2J) and symmetric algorithms by pruning the non-skyline-join tuples to get the final result set. Anuradha Awasthi *et al.* Paper [20] studied the k-dominant skyline-join queries (KSJQ) algorithm to solve the k-dominant skyline query problem in the multi-relationship complete library. Based on the proposed group division approach, Zhang *et al.* [24] proposed an efficient algorithm Skyjog, which was applicable for skyline-join on two or even more relations. For the skyline query problem of multiple data streams, Zhang *et al.* [21] proposed the Naive Parallel Sliding Window Join (NP-SWJ) and Incremental Parallel Sliding Window Join (IP-SWJ) algorithms. Bai *et al.* [9] proposed the distributed skyline-join query algorithm (DSJQ) to process skyline-join queries in distributed databases. However, none of these tasks involves incomplete databases. Alwan *et al.* [22] proposed a skyline algorithm in a non-holonomic database, but the algorithm was based on the traditional incomplete database skyline, that was, the algorithm only compared in dimensions where the data was not missing, and the returned skyline result set did not meet the user's needs.

In practical applications, the data dimensions that people care about may not exist in the database, and often relate to multi-sets queries. Therefore, on the incomplete datasets, this paper proposes efficient skyline-join query processing algorithms using crowdsourcing.

B. QUERY WITH CROWDSOURCING

For Max query, Verroios *et al.* [25] proposed the strategies for selecting the max algorithm parameters to resolve the best match with the input. They proposed strategies for selecting the max algorithm parameters to resolve the best match with the input. For top-k query, Lee *et al.* [18] proposed the CrowdK algorithm to solve the top-k problem, and the algorithm used two-stage parametric framework with two parameters, bucket, and range to reduce the cost of crowdsourcing by controlling the range of buckets. Liu *et al.* [27] combined the student's distribution estimation and the Stein's estimation with crowdsourcing to ensure the quality of crowdsourcing. Davidson *et al.* [40] studied the question of using crowd answers to evaluate max/top-k and group queries. Assuming that the probability that the crowd correctly answered each type or value question was greater than 1/2, a variable error model was proposed for the wrong answers of the crowd. For skyline, Lee *et al.* [10] proposed a crowdsourcing-based CrowdSky algorithm to minimize the cost of money, but the running time had increased.

The above researches are limited to a single relationship, and do not support multi-relational skyline queries well. This paper extends the skyline query based on crowdsourcing to make it suitable for skyline queries with missing data for multiple relationships.

C. SKYLINE QUERY OF INCOMPLETE DATASET

There are also many research results of skyline query on incomplete data [10], [12], [14], [16], [26], [41]–[49], [51], [52], and their applications are also very extensive.

The skyline query in the incomplete dataset was first proposed by Khalefa *et al.* [14]. This paper gave a definition of skyline query in an incomplete dataset, which was the dominance of the two data tuples was obtained only by comparing the attribute dimensions in which they were not missing data. However, this would destroy the transitive relationship among the data tuples, leading to the problem of circular domination between tuples. Zhang *et al.* [42] studied queries that returned top-k results on incomplete data, and proposed an extended

skyband based (ESB) algorithm and an upper bound based (UBB) algorithm to effectively reduce candidate sets.

Zhang *et al.* [16] proposed the concept of probability skyline on incomplete dataset, and proposed an efficient algorithm PISkyline to return k points with the highest skyline probabilities. Reference [42] proposed SPISkyline, SPCSkyline and SPASkyline algorithms to calculate probability skyline for incomplete data of independent, correlated and anti-correlated distribution. Kou *et al.* [28] proposed U-Skyline (uncertain skyline) query that searched for a set of tuples that had the highest probability as the skyline answer. Meeyai [12] proposed the Crowd-Enabled Skyline Queries algorithm (CEAQ), combining the dynamic crowdsourcing with heuristic technology. Compared with the probabilistic skyline, although the algorithm's query cost and running time had increased, the quality of the expected results increased significantly. Miao *et al.* [43] proposed a Bayesian query framework, and proposed an adaptive Davis-Putnam-Logemann-Loveland (DPLL) algorithm on this framework. Swidan *et al.* [50] proposed an approach for estimating the missing values of the skylines by first exploiting the available data and utilized the implicit relationships between the attributes in order to impute the missing values of the skylines. Lee *et al.* [10] proposed a crowdsourced Crowdky algorithm to solve the skyline query problem on a single relation.

But the above researches were mostly based on probability, uncertainty, and the returned results tended to be biased or were limited to a single relationship and did not support multi-relational skyline queries. This article extends the crowdsourcing skyline query to make it suitable for skyline queries with multiple relational datasets.

III. PARTIAL SKYLINE-JOIN WITH CROWDSOURCING

In this section, we first show some definitions of our research problem. Second, we present the detail process and the algorithm of the partial skyline-join with crowdsourcing.

A. PROBLEM DEFINITION

Definition 1 (Dominance): Given two data tuples t_i and t_j with d dimensions, $t_i\ (p)$ is the value of t_i on the p field. The tuple t_i governs t_j, if and only if the following conditions are holds:$\forall p(0 \le p \le d)t_i(p) \le t_j(p) \wedge \exists q(0 \le q \le d)t_i(p) < t_j(p)$, and it is denoted as $t_i \prec t_j$.

Definition 2 (Skyline Query): Dataset A has n tuples. The skyline is a set of tuples that are not dominated by any other tuples in A. We use $Sky(A)$ to represent the skyline query $Sky(A) = \{t_i | \forall t_i, t_j \in A,\ \nexists t_j \prec t_i\ \}$.

Most of the existing research works assume that skyline queries are limited to one dataset. However, this assumption is no longer valid in the Internet environment. In this case, we need to query and process data from multiple datasets, so we introduce skyline-join query as follows.

Definition 3 (Skyline-Join Query): The attribute involves a skyline query of multiple datasets $(A_1, A_2, \ldots, A_k)$, defined as skyline-join query. Let dataset $B = \{A_1 \bowtie A_2 \bowtie \ldots \bowtie A_K\}$, then skyline-join of B is $skyline(B) = \{t | t \in B, \nexists P \in B, P \prec B\}$.

Definition 4 (Incomplete Dataset): Incomplete Dataset refers the dataset which has no value or misses in some attributes.

Definition 5 (Skyline-Join Query of Incomplete Datasets With Crowdsourcing): On the datasets, the preference relationship between tuples in the missing dimension is returned through crowdsourcing, and combines the preference relationship in the dimension whose attribute value is known, and attributes with known attribute values come from a known dataset or an incomplete dataset, and then returns the result of the Skyline-join query of the incomplete datasets.

Definition 6 (Partial Skyline-Join With Crowdsourcing PSJCrowd): PSJCrowd refers that for two datasets of skyline-join query, all the attributes of one dataset are known, and only one attribute of the other dataset is unknown, where the skyline-join query only involves the unknown crowdsourcing attribute and the join attribute.

In this section, we mainly introduce the process of the first situation of Partial Skyline-Join with Crowdsourcing (PSJCrowd). First, the known dataset is filtered. Second, the incomplete dataset is filtered by two-to-two crowdsourcing comparison. Finally, an index is established based on the known dataset. The overall data is filtered by our proposed PSJCrowd algorithm to obtain the final results.

B. KNOEN DATASET FILTERING

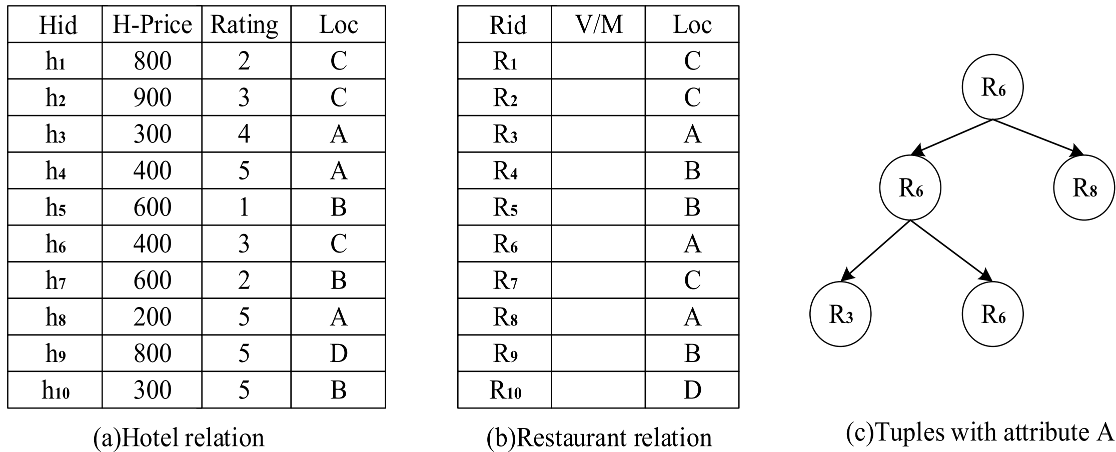
In order to quickly respond to user query and improve query efficiency, we first filter the tuples in the known dataset, using the skyline query algorithm (BNL) [1] to quickly filter out the non-skyline result tuples, and return the candidate set in the known dataset.

The filtering process of the known dataset is as follows. We first divide the tuples in the known dataset into different groups according to the join attribute, and then use the BNL algorithm to find the skyline result set in each group. Finally, return the skyline result set of all the groups.

Example 1: Figure 2 shows an example of PSJCrowd. Figure 2(a) is the hotel relation which is the known dataset and has 10 tuples. Suppose the smaller the better. We can see that the skyline results of value 'A' of attribute 'Loc' is $\{h_3, h_8\}$, and the skyline results of value 'B' of attribute 'Loc' is $\{h_5, h_{10}\}$. So, the dataset after subtraction is $\{h_1, h_3, h_5, h_6, h_8, h_9, h_{10}\}$.

C. UNKNOWN DATASET FILTERING

In order to reduce the cost of crowdsourcing, we use the method of pairwise comparison based on the tournament algorithm [26] to query crowdsourcing datasets with unknown attribute. Pairwise comparisons are performed between two tuples and the winner proceeds to the next round until the best tuple is found. When crowdsourcing on the attribute set with missing attributes, it is known through some attributes that tuples cannot dominate each other on the missing attribute set or through known attributes and partial



Hid	H-Price	Rating	Loc
h_1	800	2	C
h_2	900	3	C
h_3	300	4	A
h_4	400	5	A
h_5	600	1	B
h_6	400	3	C
h_7	600	2	B
h_8	200	5	A
h_9	800	5	D
h_{10}	300	5	B

(a)Hotel relation

Rid	V/M	Loc
R_1		C
R_2		C
R_3		A
R_4		B
R_5		B
R_6		A
R_7		C
R_8		A
R_9		B
R_{10}		D

(b)Restaurant relation

(c)Tuples with attribute A

FIGURE 2. An example of partial skyline-join with crowdsourcing (PSJCrowd).

crowdsourcing, the two tuples cannot be controlled by each other, we can stop crowdsourcing, and there is no need to crowd each missing attribute on the missing attribute set of the tuple. In this way, the preference relationship of tuples is returned in the entire missing dimension to reduce the crowdsourcing cost.

The query process is as follows. We compare the tuples of the same attribute in pairs, and the winner enters the next round. Because in this case, skyline-join query only cares about one attribute in this dataset, the skyline result is the optimal value within each attribute.

Example 2: As shown in Figure 2(b), there are 10 tuples in the restaurant relation. When we only care about the *cost-effective* (V/M) of a restaurant and the value of V/M is unknown, so we need crowdsourcing to solve this problem. For example, the tuple with attribute 'A', as shown in Figure 2(c), we first crowd (R_3, R_6). Suppose we obtain that R_6 is better than R_3, then we return R_6 and go to the next round until the final results are obtained. The same operation is performed in the attribute 'B', 'C', and 'D'. And the final result set $\{R_1, R_6, R_4, R_{10}\}$ is returned. In this step we only need 2 rounds with 6 questions to solve the problem.

D. GLOBAL DATASETS FILTERING

We separately filter the datasets to be joined. We compare the linked tuples in pairs, that is, we need to crowd between the paired tuples, which would produce huge crowdsourcing overhead. Therefore, we propose a level-preference-tree-index (LPT) with early termination conditions to reduce the cost of crowdsourcing, shorten the time delay and obtain the final skyline-join result.

To facilitate subsequent operations, reducing the cost of crowdsourcing, we now adopt a level-preference-tree-index that visualizes the preferences of tuples in dataset in order to compute skyline-join result efficiently. LPT is a kind of tree structure index. Each node in the tree represents a tuple, and the directed edge in the tree represents the dominance relationship between nodes, and the priority tree supports the transitivity of the dominant relationship between the tuples in any two layers, that is, supports cross-layer domination. After asking each question, LPT is iteratively updated, and is used for identifying the dominance relationships by checking the path between tuples. In particular, the initial virtual node is constructed for future calculations, and after each round of traversal, the node level is updated based on the virtual node. The early termination condition is added to LPT, that is, only the parent node is traversed in the previous round, and the child node can participate in the traversal in the next round. The lower tuples do not need to be traversed.

Figure 3 is an example of our LPT. We introduce a LPT based on known dataset, the dominance relationship can be represented by a directed edge between two tuples across different layers, named PSJCrowd-T. The skyline layers permit the dominance relationship between tuples in any two layers, for example, $h_3 \rightarrow h_{10}$ and $h_6 \rightarrow h_9$ in Figure 3(b). The dominance relationship via transitivity can be inferred from multiple edges.

Algorithm 1 provides the pseudo-code for filtering global data using the level-preference-tree-index (*PSJCrowd-T*). The tuples after the first-level tuple join in the level-preference-tree-index must be in the skyline-join result set (lines 1-5), and iterating from the second level of the index. If the tuple has only one parent node, crowdsourcing of the tuple after the parent node and the child node are joined (lines 6-13). We determine whether a parent node with the same join attributes dominates the same node at the same time. If so, only one parent node is kept to its edge, and all other edges are broken (lines 14-16). When the link attributes between the parent nodes are different, we check whether the tuple is dominated by two or more parent nodes, if so, if the parent nodes have been traversed, the dominant relationship between the parent nodes may be crowdsourced first (lines 17-22). Before each crowdsourcing, it is necessary to check whether the preference relationship of attribute requiring crowdsourcing already exists, and obtain it through crowdsourcing or transitivity. This node can participate in crowdsourcing only if the parent node has been facilitated in

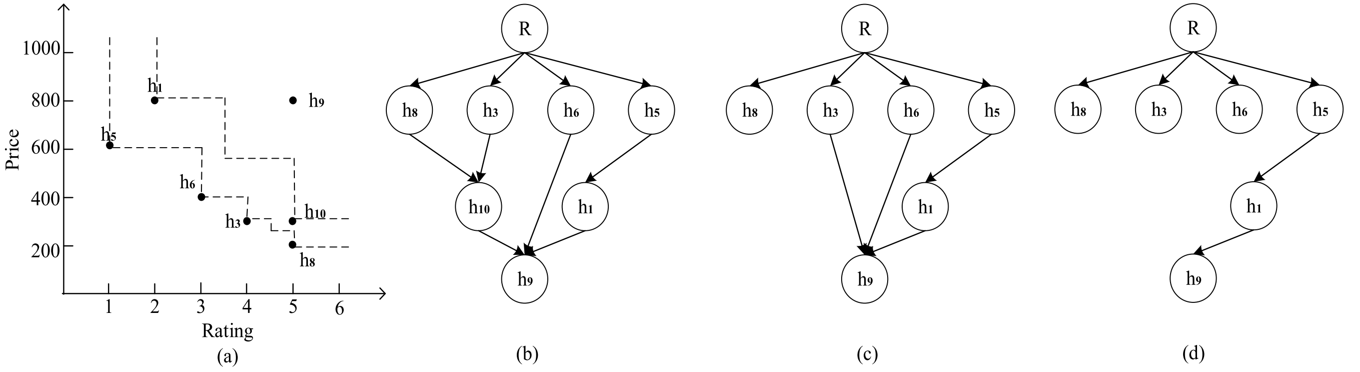


FIGURE 3. An example of level-preference-tree-index.

Algorithm 1 PSJCrowd ($H' \bowtie R'_1$)

Input: H', R_1, Index with H'
Output: Sky ($H' \bowtie R'_1$)
1: **For each** ($h_i \in H'$) **do**
2: **If** ($h_{\text{i.falg}} = 1$) **then**
3: **If** ($h_{\text{i.loc}} = r_{\text{i.loc}}$)**then**
4: $\text{h}_\text{i} \bowtie \text{r}_\text{i} \in$ Sky ($H' \bowtie R'_1$)
5: Initialize a preference tree *PSJCrowd-T* in R'_1
6: **While** ($\exists h_{\text{i.flag}} > 1$) **do**
7: **If** ($h_{\text{i.fnum}} = 1$) **then**
//$h_j \prec h_i$,$h_{i.loc} = r_{k.loc}$,$h_{j.loc} = r_{z.loc}$
8: **If** ((r_k,r_z) $\notin$*PSJCrowd-T*) **then**
9: Ask(r_k,r_z) to crowds, and update *PSJCrowd-T* with (r_k,r_z)
10: **If** ($r_z \prec r_k$) **then**
11: Remove h_i
12: **Else** $h_i \bowtie r_k \in$ Sky ($H' \bowtie R'_1$)
13: $h_{i.flag}$=1
14: **else if** ($\exists$num($h_l \prec h_i \wedge h_{l.flag}$=1)$\geq$2) **then**
//$h_j \prec h_i$, $h_m \prec h_i$,$h_{j.loc} = r_{z.loc}$,$h_{m.loc} = r_{n.loc}$
15: **If** ($h_{j.loc} = h_{m.loc}$) **then**
16: Remove edge ($h_m \rightarrow h_i$) in index
17: **else if** ($\exists$num ($h_l \prec h_i \wedge h_{l.flag}$=1)$\geq$2) **then**
18: **If** ($\nexists$(r_k,r_z)) **then**
19: Ask (r_z, r_n) to crowds, and update *PSJCrowd-T* With (r_z, r_n)
20: **If** ($r_z \prec r_n$) **then**
21: Remove edge ($h_m \rightarrow h_i$) in index
22: **else** Remove edge ($h_j \rightarrow h_i$) in index
23: **Output** Sky ($H' \bowtie R'_1$)

the previous round. Repeat each round until all the tuples have been traversed.

Example 3: Assume that the preference order among R_1, R_6, R_4, and R_{10} is $R_{10} \prec R_1 \prec R_6 \prec R_4$. We need to crowd them in pairs to get this result. According to Algorithm 1, the data of the first layer does not need to be traversed, and the tuples after the join are the skyline-join set. First, we enter the first round, because the link attributes of h_3 and h_8 are 'A', so we only crowd tuples that need to be joined to h_3, h_{10}, that is, {(R_6, R_4)} needs to be queried and checked before crowdsourcing. Whether the preference relationship of the lower tuples already exists in *PSJCrowd-T*, if it exists, directly operate the level-preference-tree-index, and if not, then crowd. In the attribute V/M, $R_6 \prec R_4$, so h_{10} is deleted, and a directed edge is added between h_3 and h_9. In addition, h_1 is dominated by h_5 only, and the preference relationship of R_1 and R_4 on V/M attributes is crowdsourced. Because $R_1 \prec R_4$ is obtained through crowdsourcing, that is, $h_1 \bowtie R_1$ belongs to the skyline-join set, so h_1 cannot be deleted, and the first round ends, as shown in Figure 3(c). In the second round, because the join properties of h_3 and h_6 are different, we need to crowdsource {(R_6, R_1)}. There are no tuples to be traversed in this round. Preference relation in Restaurant updated to Figure 4. In the third round, h_6 and h_1 have a common link attribute 'C', so the tuples R_1 and R_{10} that need to be joined to h_1 and h_9 are crowdsourced, and the traversal ends, as shown in Figure 3(d). We go through three rounds of operation and complete the filtering process with 4 questions. The skyline-join is finally identified as {($h_8 \bowtie R_6$), ($h_3 \bowtie R_6$), ($h_6 \bowtie R_1$), ($h_1 \bowtie R_1$), ($h_9 \bowtie R_{10}$)}.

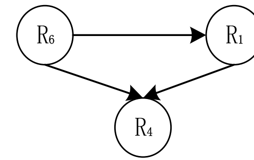


FIGURE 4. Preference relation in restaurant after two rounds.

Throughout the process, we pass 10 questions in 5 rounds, 35 fewer problems than the 45 ($10 \times 9/2$) questions in the baseline method. We greatly reduce the cost of crowdsourcing.

IV. ALL SKYLINE JOIN WITH CROWDSOURCING

In this section, we propose the all skyline-join with crowdsourcing problem. First, we give the problem definition of ASJCrowd.

Definition 7 (All Skyline-Join With Crowdsourcing, ASJCrowd): ASJCrowd problem refers that for two datasets of skyline-join query, all the attributes of one dataset are

Rid	Dis	R-Price	Quality	Loc
R1	20	80		C
R2	30	20		C
R3	20	80		A
R4	10	60		A
R5	40	10		A
R6	50	40		B
R7	80	60		B
R8	50	70		A
R9	40	40		A
R10	50	90		A
R11	50	60		C
R12	60	50		A
R13	70	70		A
R14	70	20		A
R15	80	40		B
R16	10	80		D
R17	80	90		A
R18	90	30		A
R19	90	10		A

Restaurant

FIGURE 5. Restaurant relation with unknown attributes.

known, and only one attribute of the other dataset is unknown, where the skyline-join query involves all the attributes not only the unknown crowdsourcing attribute and the join attribute.

Then, the processing course of ASJCrowd is proposed. The processing of the known dataset is the same as the first case, so the description does not repeat.

A. UNKNOWN DATASET FILTERING

First, we filter the dataset containing unknown attribute. In order to reduce the cost of crowdsourcing and reduce the time delay, we build the index based on the LPT construction method. That is, a ASJCrowd-single-level-preference-tree-index (ASJCrowd-ST) is constructed based on the preference relationship between the known attributes on the unknown dataset. The operations are the same on tuples with different join attributes.

Such as, the user needs to find a low-priced and good-star hotel, and to find a restaurant close to the hotel which is cheap and has a high quality. At the same time, the 'Quality' of the restaurant is the user's evaluation of the restaurant cost-effective and the data is missing. The Restaurant dataset becomes as shown in Figure 5. Taking the tuple with join attribute 'A' as an example, we build an index ASJCrowd-ST based on the known attributes ('Dis' and 'R-price') of the Restaurant dataset, as shown in Figure 6(b).

Algorithm 2 provides the pseudo-code for filtering data on unknown datasets. The tuples after the first-level tuple join in the level-preference-tree-index must be in the skyline-join result set (lines 1-5) and iterate from the second level of the index. If the tuple has only one parent node, the parent and child nodes are crowdsourced on unknown attributes

Algorithm 2 ASLCrowd-single(c)

Input: R_2, Index with ^k_R
Output: Sky(R_2)
1: **For each** ($r_i \in c$) **do**
2: **If** ($r_{i.flag} = 1$) **then**
4: $r_i \in$ Sky(R_2)
5: Initialize a preference tree *ASJCrowd-ST* in R^c
6: **While** ($\exists r_{i.flag} > 1$) **do**
7: **If**($r_{i.fnum} = 1$) **then**
//$r_j \prec r_i$ in R^k
8: **If** ($\nexists(r_i, r_j)$) **then**
9: Ask (r_i, r_k) to crowds, and update *ASJCrowd-ST* with (r_i,r_j)
10: **If** (r_i,r_j) **then**
11: Remove r_i
12: **Else** $r_i \in$ Sky(R_2)
13: **else if** ($\exists$ num($r_j \prec r_i \wedge r_{j.flag} = 1$)$\geq$2) **then**
14: //$r_j \prec r_i$ in ^k_R, $r_k \prec r_i$ in ^k_R
15: **If** ($\nexists(r_j, r_k)$) **then**
16: Ask (r_j,r_k) to crowds, and update *ASJCrowd-ST* with (r_j,r_k)
17: **If** ($r_j \prec r_k$ in R^c) **then**
18: Remove edge ($r_k \rightarrow r_i$) in index
19: **else** Remove edge ($r_j \rightarrow r_i$) in index
20; **Output** Sky(R_2)

(lines 6-12). If there are multiple parent nodes, as long as the parent node is traversed, the preference relationship between the parent nodes on the unknown attributes can be crowdsourced (lines 13-19). Before each crowdsourcing, it is necessary to check whether the preference relationship of the attribute requires crowdsourcing already exists, and obtain it through crowdsourcing or transitivity. This node can participate in crowdsourcing only if the parent node has been facilitated in the previous round.

Example 4: Assume that the true preference relationship between the missing attributes '*Quality*' of the tuple on the incomplete dataset is shown in Figure 6(a). According to Algorithm 2, R_4, R_9, R_{14}, and R_{19} are in the skyline-join candidate set. In the first round, $R_4 \prec R_3$ in *ASJCrowd-ST* and the preference relationship between R_3 and R_4 on the '*Quality*' attribute is missing, so $\{(R_3, R_4)\}$ is crowdsourced. Because $R_4 \prec R_3$ also exists in the '*Quality*' attribute, we delete node R_3 and add edges $R_4 \rightarrow R_5$, and $R_4 \rightarrow R_{10}$, and update *ASJCrowd-ST* at the same time. The crowdsourcing must update *ASJCrowd-ST* every time, and the latter is no longer to repeat. In addition, $R_4 \prec R_8$, $R_9 \prec R_8$ in *ASJCrowd-ST*, so $\{(R_4, R_9)\}$ is crowdsourced first. On the 'Quality' property, $R_9 \prec R_4$, so $R_4 \rightarrow R_8$ is deleted. At the same time, we crowd $\{(R_9, R_{12})$ and $\{(R_{14}, R_{19})\}$, deleting node R_{12}, joining edge $R_9 \rightarrow R_{13}$ and deleting edge $R_{19} \rightarrow R_{18}$. The round is over. In the second round, on the 'Quality' property, $R_9 \prec R_4$ can be obtained in the previous round of crowdsourcing, so delete the edge $R_4 \rightarrow R_5$. For $\{(R_9, R_5)\}$ crowdsourcing, the node R_5 is deleted in *ASJCrowd-ST*.

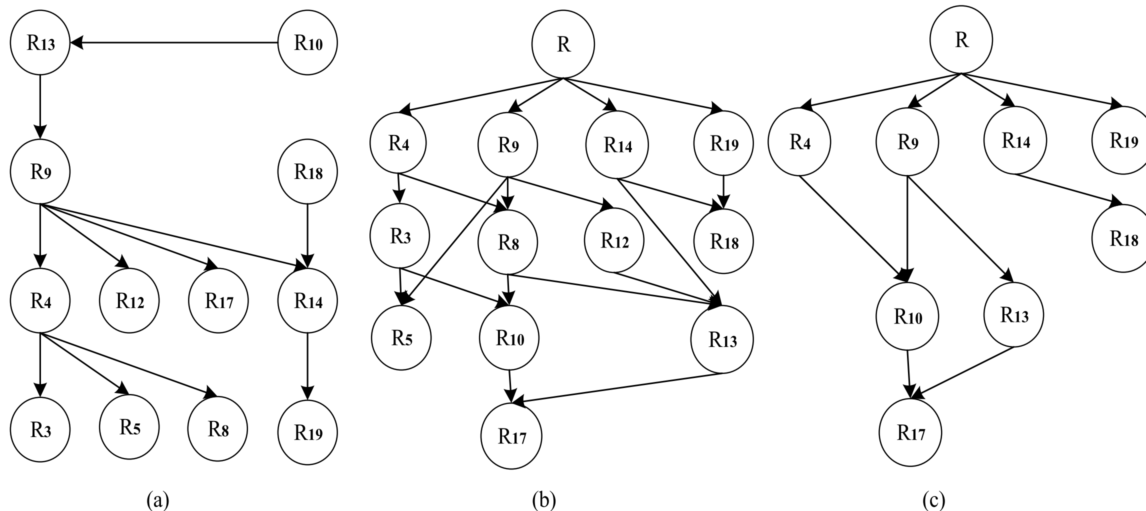


FIGURE 6. Level-preference-tree-index on tuples with attribute 'A'.

In addition, crowdsourcing $\{(R_9, R_8)\}$, node R_8 is deleted. At the same time, crowdsourcing for $\{(R_9, R_{14})\}$, $\{(R_{14}, R_{18})\}$, edge $R_{14} \rightarrow R_{13}$ is deleted. The round ends. At this time, *ASJCrowd-ST* is updated as shown in Figure 6(c), and the preference relationship of the tuples on the incomplete dataset on the 'Quality' attribute can be updated as shown in Figure 7. Crowdsourcing $\{(R_9, R_{10})\}$, $\{(R_9, R_{13})\}$ is in the third round, and the next round of crowdsourcing is $\{(R_{10}, R_{13})\}$. In the last round of crowdsourcing, $\{(R_{10}, R_{17})\}$ is crowdsourced, and the related operations are updated according to the crowdsourcing results. After updating *ASJCrowd-ST*, as shown in Figure 8, we return the result after reduction $\{R_4, R_9, R_{14}, R_{19}, R_{10}, R_{13}, R_{18}\}$.

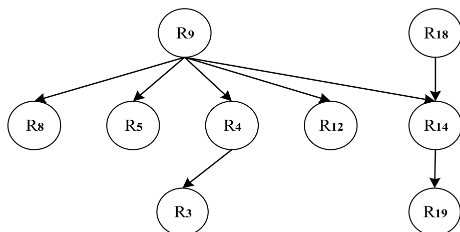


FIGURE 7. Preference relationship in restaurant relation in quality after two rounds.

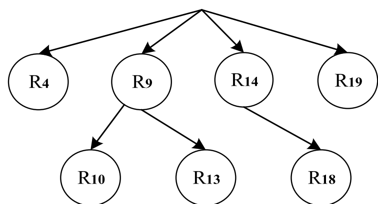


FIGURE 8. Level-preference-tree-index on tuples with attribute 'A'.

Similarly, filter the tuples with attributes 'B', 'C' and 'D'. On the 'Quality' attribute, assume $R_2 \prec R_{11}, R_{15} \prec R_6, R_{15} \prec R_7$. In the tuple containing the attribute 'B', the pairs of $\{(R_6, R_{15})\}$ and $\{(R_{15}, R_7)\}$ are crowded by two rounds, and the result $\{R_6, R_{15}\}$ is returned. In the tuple containing the attribute 'C', the traversal is completed by one round of inquiry $\{(R_2, R_{11})\}$, and the result $\{R_1, R_2\}$ is returned. That is, in this step, the operation is ended by 5 rounds of 15 questions, and the result is returned. The set is $\{R_4, R_9, R_{14}, R_{19}, R_{10}, R_{13}, R_{18}, R_6, R_{15}, R_1, R_2, R_{16}\}$.

B. GLOBAL DATASET FILTERING

In the last step of this method, the tuples in the two datasets after filtering are first joined according to the join attributes, and then a level-preference-tree-index is established to filter the global data. Using the previously mentioned method, then we build a ASJCrowd-multiple-level-preference-tree-index (ASJCrowd-MT) based on the attributes of the known dataset and the known attribute columns of the unknown dataset, and as shown in Figure 9.

Algorithm 3 provides the pseudo-code for filtering global data using the level-preference-tree-index (*ASJCrowd-MT*), then the tuple of the first layer in *ASJCrowd-MT* must be in the skyline-join result set (lines 1-5), and iterating from the second layer of the index. If the tuple has only one parent node, then determines whether the join properties of the parent and child nodes are the same. If the join properties are the same, crowdsourcing is not required. Both the parent and child nodes belong to the skyline-join set. If the join attributes are not the same, crowdsourcing the parent and child nodes on unknown attributes (lines 7-13). If the tuple is dominated by two or more tuples, first determine whether the join attributes between the parent nodes are consistent. If they are consistent, then check whether the tuples on the joined incomplete dataset are the same. If the same tuple is joined, crowdsourcing is sufficient once. If they are not the same, first check whether the preference relationship of the joined tuples on the unknown attribute already exists. If it exists, crowdsourcing is only once. If the join properties are inconsistent, the preference relationship between the parents nodes on the unknown properties can be crowdsourced first

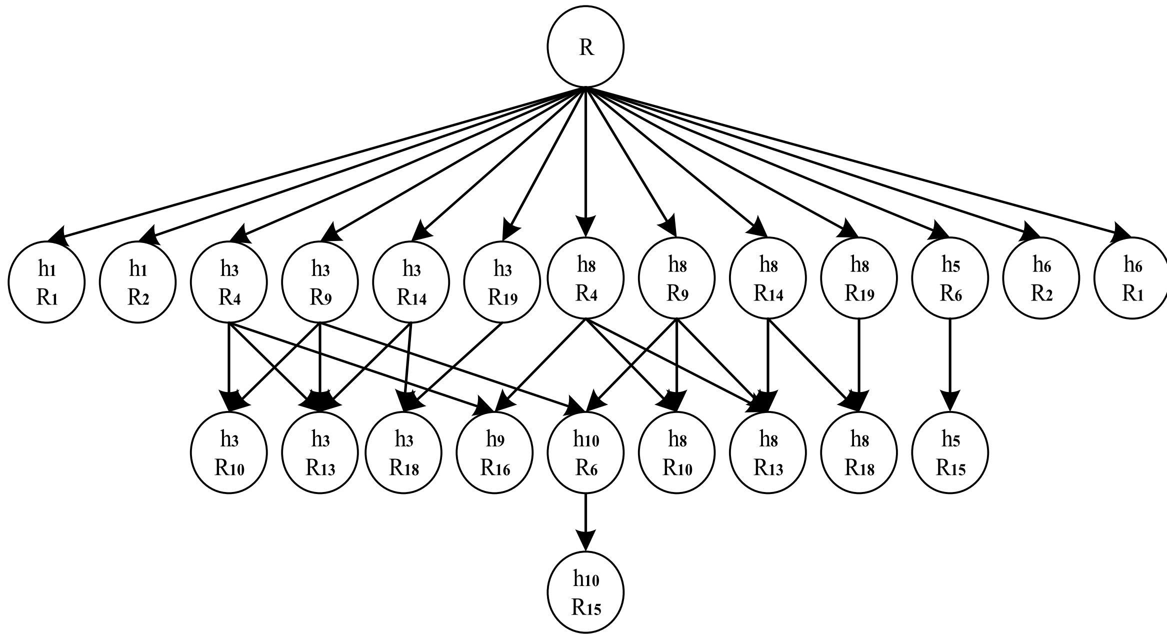


FIGURE 9. Level-preference-tree-index on all known attributes.

(lines 14-27). Before crowdsourcing, it is necessary to check whether the tuple's preference relationship on unknown attributes exists. The preference relationship can be obtained through crowdsourcing and domination transitivity. The crowd can be crowdsourced in the next round only after the parents' convenience has passed. Repeat the above operation until the convenience is completed.

Example 5: In the attribute that requires crowdsourcing, it is assumed that the true preference relationship is $R_9 \prec R_6, R_{15} \prec R_9, R_{16} \prec R_4$, and crowdsourcing is required to obtain it. According to Algorithm 3, the tuples after the join of the first layer are in the skyline-join set. The first round is traversed from the second layer. The link attributes of $h_3 \bowtie R_9$ and $h_3 \bowtie R_{10}$, $h_3 \bowtie R_{13}$ are the same, so there is no crowdsourcing. They are all in the skyline-join result set. In the index tree, $h_3 \bowtie R_9$ and $h_8 \bowtie R_9$ all dominate $h_{10} \bowtie R_6$, $h_3 \bowtie R_9$ and $h_8 \bowtie R_9$ have the same attributes, and h_3 and h_8 are joined to R_9, so we only need to set $\{(R_9, R_6)\}$ crowdsourcing. Because $R_9 \prec R_6$, we delete nodes $h_{10} \bowtie R_6$. Similarly, in the global index tree, $h_3 \bowtie R_4 \prec h_9 \bowtie R_{16}$, $h_8 \bowtie R_4 \prec h_9 \bowtie R_{16}$, so for $\{(R_4, R_{16})\}$ crowdsourcing, because $R_{16} \prec R_4$, so $h_9 \bowtie R_{16}$ also belongs to the final skyline-join set. In the second round, in the index tree, $h_3 \bowtie R_9$ and $h_8 \bowtie R_9$ both dominate $h_{10} \bowtie R_{15}$, $h_3 \bowtie R_9$ and $h_8 \bowtie R_9$ have the same join attributes. And h_3 and h_8 are joined to R_9, so only need to crowdsourcing on $\{(R_9, R_{15})\}$ on properties. The traversal ends. The remaining ones do not need crowdsourcing because of the same join properties. The tuples after the join shown in Figure 9 except for $h_{10} \bowtie R_6$ are the results of the skyline-join query. In this step, three questions are asked through two rounds to get the final result.

Throughout the process, we pass 18 questions in 7 rounds, 172 fewer problems than the 190 $(20 \times 19/2)$ questions in the baseline method of pairwise comparison. Our algorithms can greatly reduce the cost of crowdsourcing.

V. EXPERIMENTAL EVALUATION

In this section, we first introduce the running environment of the experiment, and then evaluate our algorithms on synthetic and real dataset. We demonstrate the performance of our algorithms in terms of crowdsourcing cost and time delay respectively.

A. EXPERIMENTAL SETUP

The experiments run under the Windows 10 environment, equipped with Core i5-6500 3.2GHz processor, 7200PRM hard disk and 8GB cache space. The Java compiler language is used in JDK1.8. We repeat each experiment multiple times and report average results.

On the synthetic dataset, it mainly focuses on the skyline-join query problem of the two datasets R1 and R2 after the many-to-many join operation. In this paper, R1 represents the complete dataset and R2 represents the incomplete dataset. In the experiment, the changes and default value in the relevant datasets are given in the Table 1.

In the experiment, we observe the trend of the crowdsourcing cost of the algorithms by changing the size of the synthetic datasets in the cluster; the number of known attributes of the known dataset; the number of the known attributes in the unknown dataset; the number of the unknown attributes of the unknown dataset. And we observe the trend of the time delay of the algorithms by changing the size of the synthetic dataset in the cluster and the number of known attributes on the unknown dataset. At the same time, while testing one of the attributes changing, the other attributes keep the default value.

Algorithm 3 ASLCrowd-multiple ($H' \bowtie R'_2$)

Input: H', R'_2, T_2, Index with H' and R^k
Output: Sky ($H' \bowtie R'_2$)
1: **For each** ($h_i \in H'$) **do**
2: **If** ($h_{i.flag}$ =1) **then**
3: **If** ($h_{i.loc} = r_{j.loc}$) **then**
4: $h_i \bowtie r_i \in$ Sky ($H' \bowtie R'_2$)
5: Initialize a preference tree *ASJCrowd-MT* in R^c
6: **While** ($\exists h_{i.flag} > 1$) **do**
7: **If** ($h_{i.fnum}$ =1) **then**
//$h_j \bowtie r_z \prec h_i \bowtie r_k \in$ index, $h_{i.loc} = r_{k.loc}, h_{j.loc} = r_{z.loc}$
8: **If** ($\nexists(r_k, r_z)$ in R^c)**then**
9: Ask (r_k, r_z) to crowds, and update *ASJCrowd-MT* with (r_k,r_z)
10: **If** ($r_z \prec r_k$) **then**
11: Remove $h_i \bowtie r_k$
12: **else** $h_i \bowtie r_k \in$ Sky($H' \bowtie R'_2$)
13: $h_{i.flag} = 1$
14: **else if** ($\exists$ num($h_l \prec h_i \wedge h_{l.flag}$ =1)$\geq$2)**then**
// ($h_j \bowtie r_z \prec h_i \bowtie r_k$, $h_m \bowtie r_n \prec h_i \bowtie r_k$) $\in$ index,
$h_{i.loc} = r_{k.loc}, h_{j.loc} = r_{z.loc}$,
$h_{m.loc} = r_{n.loc}$
15: **If** ($h_{j.loc} = h_{m.loc}$) **then**
16: **If** ($r_z = r_n$) **then**
17: Remove edge ($h_m \bowtie r_n \prec h_i \bowtie r_k$) in index
18: **else if** ($\nexists(r_z = r_n)$ in R^c) **then**
19: Ask (r_z, r_n) to crowds, and update *ASJCrowd-MT* with (r_z, r_n)
20: **If** ($r_z \prec r_k$) **then**
21: Remove edge($h_m \bowtie r_n \rightarrow h_i \bowtie r_k$) in index
22: **else** Remove edge ($h_j \bowtie r_z \rightarrow h_i \bowtie r_k$) in index
23: **else if** ($\nexists(r_z,r_n)$ in R^c)**then**
24: Ask (r_z,r_n) to crowds, and update *ASJCrowd-MT* With (r_z, r_n)
25: **If** ($r_z \prec r_n$) **then**
26: Remove edge($h_m \bowtie r_n \rightarrow h_i \bowtie r_k$) in index
27: **else** Remove edge ($h_j \bowtie r_z \rightarrow h_i \bowtie r_k$) in index
28: **Output** Sky ($H' \bowtie R'_2$)

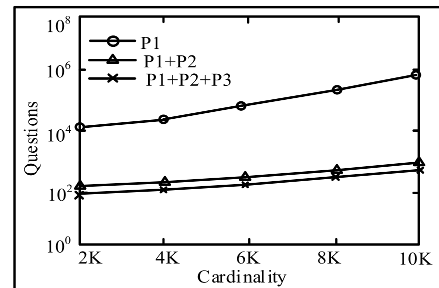
The crowdsourcing-based skyline-join query algorithm proposed in this paper is called PSJCrowd algorithm and ASJCrowd algorithm. In terms of crowdsourcing costs, for the synthetic datasets, the comparison algorithm we chose is based on the baseline method of pairwise comparison and itself. In terms of time delay, the comparison algorithm we chose is a scoring-based with unary questions baseline method that asks only one question at a time and itself. The experiment analyzes the changes of crowdsourcing cost and time delay when the algorithm runs from two aspects: the change of the attribute and the change of the database size.

TABLE 1. Parameter settings over synthetic datasets.

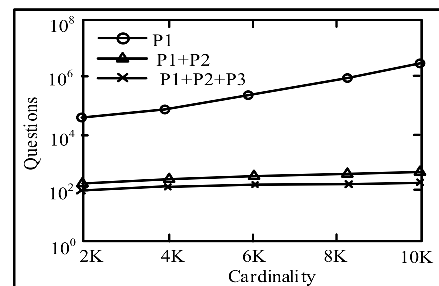
Parameter	Value	Default
Cardinality n	2K, 4K, 6K, 8K, 10K	4K
|Attribute of R1|	1,2, 3, 4	2
|Known attribute of R2|	1, 2, 3	2
|Unknown attribute of R2|	1, 2, 3	1

B. EXPERIMENTAL RESULTS

The Cost of Crowdsourcing: When paying a fixed amount of reward for each question, the crowdsourcing cost is proportional to the number of questions asked to the crowd. We use the number of asked questions to measure crowdsourcing costs. It is divided into three phases: P1 (filtering only on R1 dataset), P2 (filtering only on R2 dataset), and P3 (global filtering). The preference relationship of unknown attributes between tuples is obtained by pairwise comparison, and it is compared with this article as a baseline method. The filtering operation on the known dataset does not affect the number of crowdsourcing, so the crowdsourcing cost of the baseline method is the same as the cost of performing only the P1 stage.



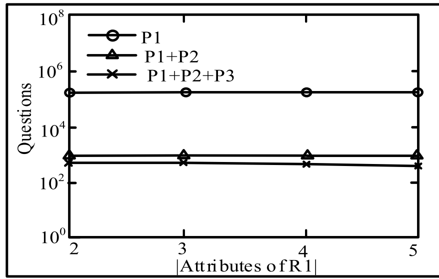
(a) Varying Cardinality of PSJCrowd



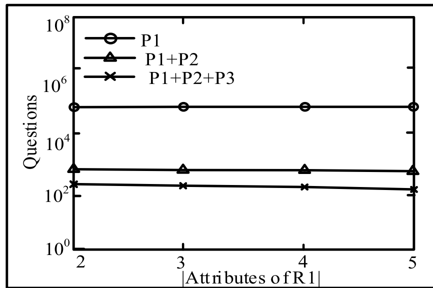
(b) Varying Cardinality of ASJCrowd

FIGURE 10. The influence of change of cardinality.

Figure 10 depicts the number of questions as the size of the dataset in both cases. Figure 10(a) and Figure 10(b) describe the number of questions based on different cardinalities in PSJCrowd and ASJCrowd. It is clear that P1+P2+P3 minimizes the number of problems with all parameter settings. As the basic number increases, the baseline method increases exponentially, and the effect of P1+P2+P3 is

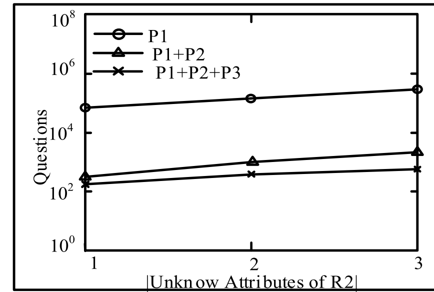


(a) Attributs of R1 in PSJCrowd

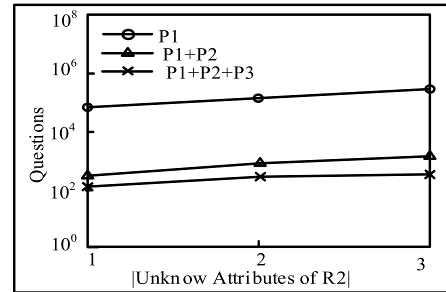


(b) Attributs of R1 in ASJCrowd

FIGURE 11. The influence of change of attributes of R1.



(a) Unknow Attributes of R2 of PSJCrowd



(b) Unknow Attributes of R2 of ASJCrowd

FIGURE 12. The influence of change of unknown attributes of R2.

more pronounced, decreasing by two orders of magnitude. In the case that unknown attributes are included, because the dominance relationship between tuples is more complicated, P3 reduces more problems, making the gap between P1+P2 and P1+P2+P3 larger.

Figure 11 shows the trend of the number of problems in two cases with the number of attributes of the known dataset. Figure 11(a) and Figure 11(b) describe the number of problems when the numbers of attributes of the known dataset are different in PSJCrowd and ASJCrowd. Although the baseline shows a constant performance regardless of the number of attributes, our construction method reduces the number of problems as the number of attributes of a known dataset increases. This is because the dominating relationship between tuples decreases as the number of attributes of known dataset increases. We find that P1 does not affect the number of problems with crowdsourcing and P1+P2+P3 minimizes the number of questions.

Figure 12 shows the trend of the number of problems in two cases with the number of known attributes of unknown dataset. Figure12(a) and Figure12(b) describe the number of problems when the numbers of unknown attributes in the unknown dataset are different in PSJCrowd and ASJCrowd. It is easy to see that P1+P2+P3 is still very effective in reducing the number of problems. When the number of unknown attributes exceeds one, we can use the looping problem to reduce the crowdsourced problem attributes. The baseline method obtains the attribute preferences between tuples by asking multiple questions at one time, so as the number of unknown attributes increases, the number of questions in the baseline method increases multiples. While the number of questions in P1+P2 and P1+P2+P3 increases slower, and the optimization effect of P3 is also more and more obvious.

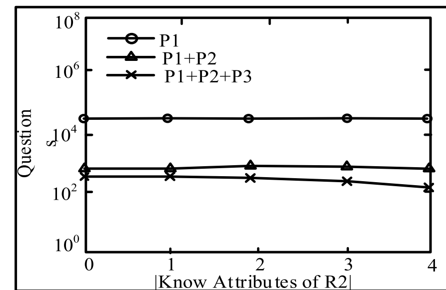
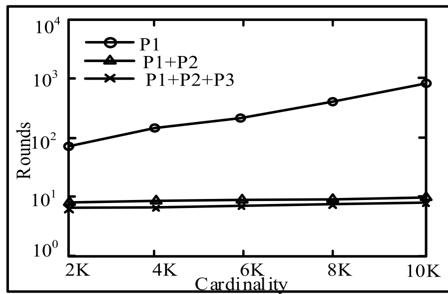


FIGURE 13. Comparisons on the number of questions over varying number of attributes in R2.

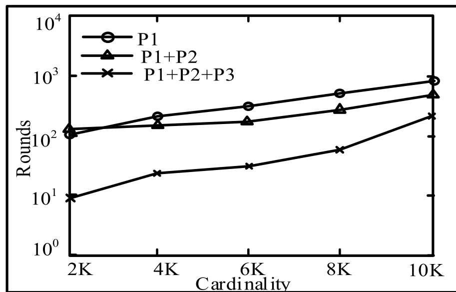
Figure 13 describes the number of questions when the number of known attributes of the unknown dataset is different. Similar to the situation described in Figure 11, as the number of known attributes of the unknown dataset increases, our construction method reduces the number of problems. And with the increase of attributes all the time, the effect of P1+P2+P3 is more obvious.

To measure the latency, we use the total number of rounds of execution of the algorithm. The baseline method is based on a scoring system that asks only one question at a time. Figure 14 depicts the trend of time delay as the size of the dataset in both cases. Figure 14(a) describes the number of rounds in PSJCrowd when the dataset size changes. Compared with the method of crowdsourcing once one problem, the number of rounds of P1+P2+P3 is reduced by two orders of magnitude. Figure 14(b) describes the number of rounds in ASJCrowd when the dataset size changes. Comparing to the baseline method and P2, the impact of P1+P2+P3 is more pronounced.

Figure 15 describes the number of rounds when the numbers of known attributes in the unknown dataset are different.



(a) Varying Cardinality



(b) Varying Cardinality

FIGURE 14. Comparisons on the number of rounds over varying cardinality.

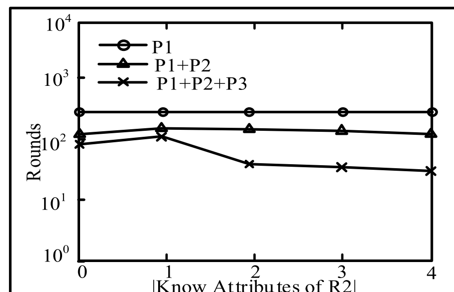


FIGURE 15. Comparisons on the number of rounds over varying number of attributes in R2.

The experimental results show that as the number of known attributes increases, the degree of parallelism is higher. When the number known attribute is larger than two, the optimization effect of P2+P3 is more obvious. The experimental results show that our proposed algorithm has high accuracy at the cost of crowdsourcing and reduces the time latency.

Figure 16 describes the impact of the ASJCrowd-single algorithm and CrowdSky algorithm on crowdsourced latency in incomplete data sets. Compared with the CrowdSky algorithm, when there are multiple nodes dominating the same child node, all the parent nodes need not be visited, as long as the parent section is traversed, the ASJCrowd-single algorithm can compare the preference relationship of the parent node on the missing attributes in advance, reducing the height of the ASJCrowd-single index, so the ASJCrowd-single algorithm is more effective in reducing crowdsourcing delay. Experimental results also show that the ASJCrowd-single algorithm is more effective than the CrowdSky [10] algorithm in reducing crowdsourcing delay, and as the data base increases, the optimization effect of the ASJCrowd-single algorithm is more obvious.

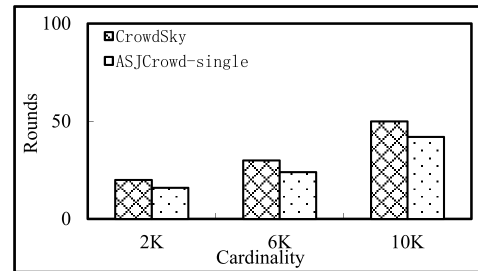


FIGURE 16. Comparison of the number of crowdsourcing rounds on different data sets.

C. EXPERIMENTAL RESULTS OF REAL-LIFE DATASETS

We use two real datasets to validate our proposed PSJCrowd and ASJCrowd algorithms. We select two places in Beijing (https://bj.meituan.com/) and Shenyang (https://sy.meituan.com/), from the website Meituan. Shenyang and Beijing are represented by S and B, respectively, and B1 and B2 represent the crowdsourcing situation of PSJCrowd and ASJCrowd on the Beijing dataset. Similarly, S1 and S2 represent the crowdsourcing situation of PSJCrowd and ASJCrowd on the Shenyang dataset. Select 500 pieces of data (10 hotels and 50 restaurants in each place). The join properties are different areas of two places. Amazon Mechanical Turk (AMT) is a well-known crowdsourcing platform and we set the cost of each crowdsourcing problem at 0.02$.

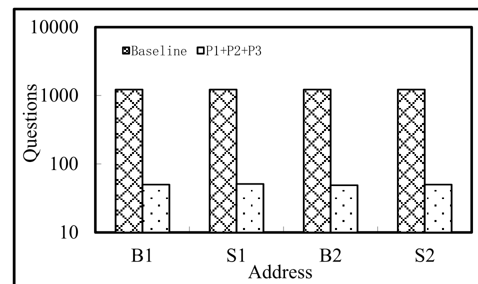


FIGURE 17. Comparisons on the number of questions over varying datasets.

Figure 17 compares the crowdsourcing cost between Baseline and PSJCrowd and ASJCrowd. It is worth noting that the crowdsourcing cost of PSJCrowd and ASJCrowd is 1/20 of the baseline method. Because the base of the real dataset is relatively small, the gap between Baseline and algorithms that we proposed is smaller than the gap on the synthetic dataset. The Baseline method requires more than 1,000 questions, while PSJCrowd and ASJCrowd only require more than 50 questions.

Figure 18 compares the crowdsourcing time delay between Baseline and PSJCrowd and ASJCrowd. The Baseline method requires 50 rounds, while PSJCrowd and ASJCrowd only need about ten rounds. The time delay of PSJCrowd and ASJCrowd is 1/5 of the baseline method. In the index, we use the preference relationship between tuples, and crowd as many problems as possible in each round, so the number of rounds of crowdsourcing is greatly reduced.

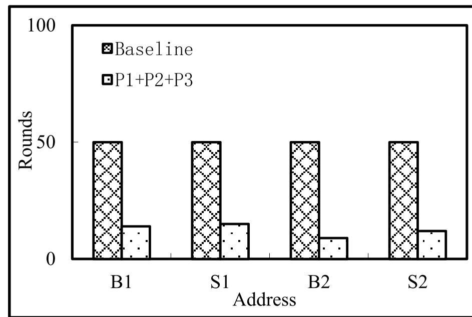


FIGURE 18. Comparisons on the number of rounds over varying datasets.

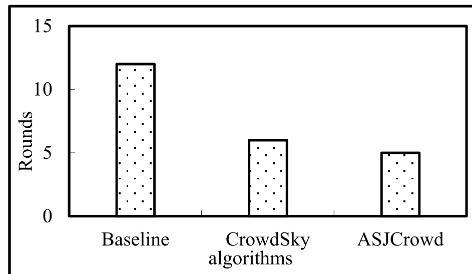


FIGURE 19. Comparisons on the number of rounds over varying algorithms.

Figure 19 describes the differences between the three algorithms in rounds in incomplete dataset. Because we research on crowdsourcing-based skyline-join query algorithms of the incomplete dataset, we compare ASJCrowd-single with crowdsourced-based skyline query algorithms (CrowdSky [10]) on a single dataset. We do experiments on tuples with a single attribute in 50 restaurants. We can see that CrowdSky and ASJCrowd-single are effective in reducing time round. Compared with the CrowdSky algorithm on a single dataset, the ASJCrowd-single algorithm does not require all the parent nodes of a child node to be traversed before crowdsourcing the preference relationship of the parent node on the missing attribute, so the ASJCrowd-single algorithm has less rounds of crowdsourcing than CrowdSky algorithm.

VI. CONCLUSION

In this paper, we study the problem of using crowdsourcing to compute skyline-join queries. Specifically, for different situations of missing data, we propose three algorithms to construct the level-preference-tree-index structure by using the dominating relationship between tuples, and trimming and filtering according to different strategies twice to reduce crowdsourcing cost and time delay. Our experimental results show that our proposed algorithm has high accuracy and meets the expectations of the public. The query results have practical significance in practical applications.

REFERENCES

[1] S. Borzsony, D. Kossmann, and K. Stocker, "The skyline operator," in *Proc. 17th Int. Conf. Data Eng.*, Heidelberg, Germany, 2001, pp. 421–430.

[2] C. L. Hwang and A. S. M. Masud, *Multiple Objective Decision Making Methods and Applications: A State-of-the-Art Survey*, vol. 164. Heidelberg, Germany: Springer, 2012.

[3] L. Dong, G. Liu, X. Cui, and T. Li, "G-skyline query over data stream in wireless sensor network," *Wireless Netw.*, vol. 26, no. 1, pp. 129–144, Jan. 2020.

[4] F. N. Afrati, P. Koutris, D. Suciu, and J. D. Ullman, "Parallel skyline queries," in *Proc. 15th Int. Conf. Database Theory (ICDT)*, Berlin, Germany, 2012, pp. 274–284.

[5] L.-Q. Pan, J.-Z. Li, and J.-Z. Luo, "Approximate skyline query processing algorithm in wireless sensor networks," *J. Softw.*, vol. 21, no. 5, pp. 1020–1030, May 2010.

[6] M. M. Müller, O. J. N. Bertrand, D. Differt, and M. Egelhaaf, "The problem of home choice in skyline-based homing," *PLoS ONE*, vol. 13, no. 3, Mar. 2018, Art. no. e0194070. [Online]. Available: http://www.journals.plos.org/plosone/article?id=10.1371/journal.pone.0194070

[7] M. Navascués, R. Leblois, and C. Burgarella, "Demographic inference through approximate-Bayesian-computation skyline plots," *PeerJ*, vol. 5, no. 11, p. e3530, Jul. 2017. [Online]. Available: http://www.peerj.com/articles/3530

[8] T. Emrich, M. Franzke, N. Mamoulis, M. Renz, and A. Züfle, "Geo-social skyline queries," in *Proc. 19th Int. Conf. Database Syst. Adv. Appl.*, Bali, Indonesia, Apr. 2014, pp. 77–91.

[9] M. Bai, J. Xin, G. Wang, R. Zimmermann, and X. Wang, "Skyline-join query processing in distributed databases," *Frontiers Comput. Sci.*, vol. 10, no. 2, pp. 330–352, Apr. 2016.

[10] J. Lee, D. Lee, and S.-W. Kim, "CrowdSky: Skyline computation with crowdsourcing," in *Proc. 19th Int. Conf. Extending Database Technol.*, Bordeaux, France, Mar. 2016, pp. 125–136.

[11] J. Song, Y. Zhang, Y. Bao, and G. Yu, "Probery: A probability-based incomplete query optimization for big data," *CoRR*, vol. abs/1901.00113, Jan. 2019.

[12] S. Meeyai, "Logistic regression with missing data: A comparison of handling methods, and effects of percent missing values," *J. Traffic Logistics Eng.*, vol. 4, no. 2, pp. 128–134, 2016.

[13] L. Zou and L. Chen, "Dominant graph: An efficient indexing structure to answer top-k queries," in *Proc. IEEE 24th Int. Conf. Data Eng.*, Cancún, Mexico, Apr. 2008, pp. 536–545.

[14] M. E. Khalefa, M. F. Mokbel, and J. J. Levandoski, "Skyline query processing for incomplete data," in *Proc. IEEE 24th Int. Conf. Data Eng.*, Los Alamitos, CA, USA, Apr. 2008, pp. 556–565.

[15] R. Bharuka and P. S. Kumar, "Finding skylines for incomplete data," in *Proc. 24th Australas Database Conf., Austral. Comput. Soc. Aust.*, Melbourne, VIC, Australia, Jan./Feb. 2013, pp. 109–117.

[16] K. Zhang, H. Gao, X. Han, Z. Cai, and J. Li, "Probabilistic skyline on incomplete data," in *Proc. ACM Conf. Inf. Knowl. Manage.*, Singapore, Nov. 2017, pp. 427–436.

[17] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom, "CrowdScreen: Algorithms for filtering data with humans," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, Scottsdale, AZ, USA, 2012, pp. 361–372.

[18] J. Lee, D. Lee, and S.-W. Hwang, "CrowdK: Answering top-k queries with crowdsourcing," *Inf. Sci.*, vol. 399, pp. 98–120, Aug. 2017.

[19] J. Wang, T. Kraska, M. J. Franklin, and J. Feng, "CrowdER: Crowdsourcing entity resolution," *Inf. Sci.*, vol. 5, no. 11, pp. 1483–1494, Jul. 2012.

[20] A. Awasthi, A. Bhattacharya, S. Gupta, and U. K. Singh, "K-dominant skyline join queries: Extending the join paradigm to k-dominant skylines," in *Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE)*, San Diego, CA, USA, Apr. 2017, pp. 99–102.

[21] J. Zhang, J. Gu, S. Cheng, B. Li, W. Wang, and D. Meng, "Efficient algorithms of parallel Skyline join over data streams," in *Proc. Int. Conf. Algorithms Archit. Parallel Process.*, Guangzhou, China, 2018, pp. 184–199.

[22] A. A. Alwan, H. Ibrahim, N. I. Udzir, and F. Sidi, "Processing skyline queries in incomplete distributed databases," *J. Intell. Inf. Syst.*, vol. 48, no. 2, pp. 1–22, 2016.

[23] A. Vlachou, C. Doulkeridis, and N. Polyzotis, "Skyline query processing over joins," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, Athens, Greece, 2011, pp. 73–84.

[24] J. Zhang, Z. Lin, B. Li, W. Wang, and D. Meng, "Skyline join query processing over multiple relations," in *Proc. DASFAA Workshops*, Dallas, TX, USA, 2016, pp. 353–361.

[25] V. Verroios, P. Lofgren, and H. Garcia-Molina, "TDP: An optimal-latency budget allocation strategy for crowdsourced MAXIMUM operations," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Melbourne, VIC, Australia, May 2015, pp. 1047–1062.

[26] P. Venetis, H. Garcia-Molina, K. Huang, and N. Polyzotis, "Max algorithms in crowdsourcing environments," in *Proc. 21st Int. Conf. World Wide Web (WWW)*, Lyon, France, Apr. 2012, pp. 989–998.

[27] X. Liu, D.-N. Yang, M. Ye, and W.-C. Lee, "U-skyline: A new skyline query for uncertain databases," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 4, pp. 945–960, Apr. 2013.

[28] N. M. Kou, Y. Li, H. Wang, L. Hou U, and Z. Gong, "Crowdsourced top-k queries by confidence-aware pairwise judgments," in *Proc. ACM Int. Conf. Manage. Data*, Sacramento, CA, USA, May 2017, pp. 1415–1430.

[29] M. Nagendra and K. S. Candan, "Efficient processing of skyline-join queries over multiple data sources," *ACM Trans. Database Syst.*, vol. 40, no. 2, pp. 1–46, Jun. 2015.

[30] E. Simperl, "Semantic crowdsourcing," in *Encyclopedia of Database Systems*. New York, NY, USA: Springer, 2017, doi: 10.1007/978-1-4899-7993-3_80606-1.

[31] G. Hollis, "Scoring best-worst data in unbalanced many-item designs, with applications to crowdsourcing semantic judgments," *Behav. Res. Methods*, vol. 50, no. 2, pp. 711–729, Apr. 2018.

[32] W. Chen, Z. Zhao, X. Wang, and W. Ng, "Crowdsourced query processing on microblogs," in *Proc. Int. Conf. Database Syst. Adv. Appl.* Cham, Switzerland: Springer, 2016, doi: 10.1007/978-3-319-32025-0_2.

[33] Q. Deng, Y. Liu, H. Zhang, X. Deng, and Y. Ma, "A new crowdsourcing model to assess disaster using microblog data in typhoon Haiyan," *Natural Hazards*, vol. 84, no. 2, pp. 1241–1256, Nov. 2016.

[34] J. A. Redi, T. Hoßfeld, P. Korshunov, F. Mazza, I. Povoa, and C. Keimel, "Crowdsourcing-based multimedia subjective evaluations: A case study on image recognizability and aesthetic appeal," in *Proc. 2nd ACM Int. Workshop Crowdsourcing Multimedia (CrowdMM)*, 2013, pp. 29–34.

[35] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin, "CrowdDB: Answering queries with crowdsourcing," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, Athens, Greece, 2011, pp. 61–72.

[36] M. K. El, C. Lofi, and W. T. Balke, "Crowdsourcing for query processing on Web data: A case study on the skyline operator," *J. Comput. Inf. Technol.*, vol. 23, no. 1, pp. 43–60, 2015.

[37] C. Lofi, K. El Maarry, and W.-T. Balke, "Skyline queries in crowd-enabled databases," in *Proc. 16th Int. Conf. Extending Database Technol. (EDBT)*, Genoa, Italy, 2013, pp. 465–476.

[38] C. Lofi, M. K. El, and W. T. Balke, "Skyline queries over incomplete data-error models for focused crowd-sourcing," *Proc. 32nd Int. Conf. Conceptual Modeling*, Hong Kong, 2013, pp. 298–312.

[39] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng, "Leveraging transitive relations for crowdsourced joins," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA, 2013, pp. 229–240.

[40] S. B. Davidson, S. Khanna, T. Milo, and S. Roy, "Using the crowd for top-k and group-by queries," in *Proc. 16th Int. Conf. Database Theory (ICDT)*, Pittsburgh, PA, USA, 2013, pp. 225–236.

[41] X. Miao, Y. Gao, B. Zheng, G. Chen, and H. Cui, "Top-k dominating queries on incomplete data," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 1, pp. 252–266, Jan. 2016.

[42] K. Zhang, H. Gao, X. Han, Z. Cai, and J. Li, "Modeling and computing probabilistic skyline on incomplete data," *IEEE Trans. Knowl. Data Eng.*, vol. 32, no. 7, pp. 1405–1418, Jul. 2020.

[43] X. Miao, Y. Gao, S. Guo, L. Chen, J. Yin, and Q. Li, "Answering skyline queries over incomplete data with crowdsourcing," *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 4, pp. 1360–1374, Apr. 2021.

[44] G. B. Dehaki, H. Ibrahim, N. I. Udzir, F. Sidi, and A. A. Alwan, "Efficient skyline processing algorithm over dynamic and incomplete database," in *Proc. 20th Int. Conf. Inf. Integr. Web-Appl. Services*, Yogyakarta, Indonesia, Nov. 2018, pp. 190–199.

[45] Y. Gao, X. Miao, H. Cui, G. Chen, and Q. Li, "Processing k-skyband, constrained skyline, and group-by skyline queries on incomplete data," *Expert Syst. Appl.*, vol. 41, no. 10, pp. 4959–4974, Aug. 2014.

[46] Y. Gulzar, A. A. Alwan, R. M. Abdullah, Q. Xin, and M. B. Swidan, "SCSA: Evaluating skyline queries in incomplete data," *Int. J. Speech Technol.*, vol. 49, no. 5, pp. 1636–1657, May 2019.

[47] Y. Zeng, K. Li, S. Yu, Y. Zhou, and K. Li, "Parallel and progressive approaches for skyline query over probabilistic incomplete database," *IEEE Access*, vol. 6, pp. 13289–13301, 2018.

[48] K. Zhang, H. Gao, H. Wang, and J. Li, "ISSA: Efficient skyline computation for incomplete data," in *Proc. Int. Conf. Database Syst. Adv. Appl.*, Suzhou, China, 2016, pp. 321–328.

[49] Y. Gulzar, A. A. Alwan, and S. Turaev, "Optimizing skyline query processing in incomplete data," *IEEE Access*, vol. 7, pp. 178121–178138, 2019.

[50] M. B. Swidan, A. A. Alwan, S. Turaev, H. Ibrahim, A. Z. Abualkishik, and Y. Gulzar, "Skyline queries computation on crowdsourced-enabled incomplete database," *IEEE Access*, vol. 8, pp. 106660–106689, 2020.

[51] X. Miao, Y. Gao, G. Chen, and T. Zhang, "k-dominant skyline queries on incomplete data," *Inf. Sci.*, vol. 367, pp. 990–1011, Nov. 2016.

[52] X. Miao, Y. Gao, L. Zhou, W. Wang, and Q. Li, "Optimizing quality for probabilistic skyline computation and probabilistic similarity search," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 9, pp. 1741–1755, Sep. 2018.

LINLIN DING received the M.Sc. and Ph.D. degrees in computer science and technology from Northeastern University, Shenyang, China, in July 2008 and 2013, respectively. She is currently an Associate Professor with the School of Information, Liaoning University, Shenyang. She published more than 40 research articles in the international conference proceedings and journals. Her research interests include big data management, uncertain data management, high-dimensional data, and distributed data management.

XIAO ZHANG is currently pursuing the master's degree with the School of Information, Liaoning University, China. Her research interests include big data management and skyline query.

HANLIN ZHANG is currently pursuing the Ph.D. degree with the School of Information, Liaoning University, China. His research interests include big data management and graph data management.

LIANG LIU is currently pursuing the bachelor's degree in software engineering with the School of Information, Liaoning University, China.

BAOYAN SONG received the B.Sc., M.Sc., and Ph.D. degrees in computer science from Northeastern University, Shenyang, China, in 1988, 1996, and 2002, respectively. She is currently a Full Professor with the School of Information, Liaoning University, China. She published more than 100 research articles in the international conference proceedings and journals. Her research interests include big data management, data stream management, and graph data management.