# Towards Query Pricing on Incomplete Data

Miao, Xiaoye; Gao, Yunjun; Chen, Lu; Peng, Huanhuan; Yin, Jianwei; Li, Qing

*Published in:*
I E E E Transactions on Knowledge & Data Engineering

*DOI (link to publication from Publisher):*
10.1109/TKDE.2020.3026031

*Publication date:*
2022

*Document Version*
Accepted author manuscript, peer reviewed version

# Towards Query Pricing on Incomplete Data

Xiaoye Miao, *Member, IEEE,* Yunjun Gao, *Member, IEEE,* Lu Chen, Huanhuan Peng, Jianwei Yin, Qing Li, *Senior Member, IEEE,*

**Abstract**—Data have significant economic or social value in many application fields including science, business, governance, etc. This naturally leads to the emergence of many data markets such as GBDEx and YoueData. As a result, the data trade through data markets has started to receive attentions from both industry and academia. During the data buying and selling, how to price the data is an indispensable problem. However, pricing incomplete data is more challenging, even though incomplete data exist pervasively in a vast lot of real-life scenarios. In this paper, we attempt to explore the *pricing problem for queries over incomplete data*. We propose a sophisticated pricing mechanism, termed as iDBPricer, which takes a series of essential factors into consideration, including the *data contribution/usage*, *data completeness*, and *query quality*. We present two novel price functions, namely, the usage, and completeness-aware price function (*UCA price* for short) and the quality, usage, and completeness-aware price function (*QUCA price* for short). Moreover, we develop efficient algorithms for deriving the query prices. Extensive experiments using both real and benchmark datasets demonstrate iDBPricer is of excellent performance in terms of effectiveness and scalability, compared with the state-of-the-art price functions.

**Index Terms**—Data trade, Data pricing, Incomplete data, Query quality

## 1 Introduction

With the high-speed development of information technology and the growing number of data generation sources, the data resource plays an increasingly important role in real life. Data are transforming science, business, and governance by enabling data-driven applications. The data usually have significant economic or social value in many application fields. As a consequence, many data markets are emerging in the past few years, which have the goals of completely enabling the data circulation and effectively solving the data isolation issue, so as to facilitate the openness and sharing of global data resources [1]. For example, GBDEx [2] possesses 225 high-quality data sources and more than 4,000 tradable data products, which has made 150PB trade volume involving more than thirty application fields. GNIP [3] aggregates and sells social media data from Twitter before the general data protection regulation (GDPR). Xignite [4] vends real-time financial data. Here [5] trades tracking and positioning data for location-based advertising. Factual [6] enables location data to power innovation in product development, mobile marketing, and real world analytics.

As depicted in Figure 1, the data market acts as an intermediator for data trade, where the data owner (i.e., seller) gets the monetary rewards, and the data consumer (i.e., buyer) obtains an amount of beneficial data. Accordingly, the financial (resp. data) asset flows efficiently from the data consumer (resp. owner) to the data owner (resp. consumer). During data trade, how to price the data is an indispensable problem, which is closely related to various factors, such as data valuation, data quality, etc. An incentive pricing mechanism benefits the healthy development of data markets.

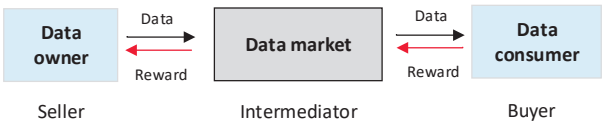



Fig. 1. Data trade on markets

Incomplete data are ubiquitous due to many reasons such as data loss, privacy preservation, instable sensor networks, etc. As a result, querying incomplete data has been extensively explored [7], [8], [9], [10], [11], [12]. In fact, many data sets sold on data markets are of incompleteness more or less, which makes the pricing problem more challenging. For instance, YoueData adopts the incompleteness degree as a factor of rating the sold data. The relatively poor quality of the data (with missing information) remarkably decreases the buyer's utility on the query, and thereby, this query should be assigned to a lower price. To this end, in this paper, we take an initial step towards selling and buying the data sets struggling with incompleteness, via exploring the *pricing problem for queries over incomplete data*.

The existing commercial data markets adopt either the traditional *subscription* mode to price data (or services), or coarse-grained pricing methods (e.g., the usage-based pricing where the price only relies on the number of data records), or the methods of pricing and selling data in *bundle* (a.k.a. the package pricing), or the *bargaining* pricing mode, such as GBDEx. These methods do not consider the effect of data incompleteness on the data price. On the other hand, some modern pricing models have been proposed in

- *X. Miao and H. Peng are with the Center for Data Science, Zhejiang University, 866 Yuhangtang Road, Hangzhou 310058, P. R. China. E-mail: {miaoxy, hhpeng}@zju.edu.cn.*
- *Y. Gao (Corresponding Author) is with the College of Computer Science, Zhejiang University, 38 Zheda Road, Hangzhou 310027, P. R. China. E-mail: gaoyj@zju.edu.cn.*
- *L. Chen is with the Department of Computer Science, Aalborg University, Aalborg, Denmark. E-mail: luchen@cs.aau.dk.*
- *J. Yin is with the Center for Data Science, the College of Computer Science, Zhejiang University, 866 Yuhangtang Road, Hangzhou 310058, P. R. China. E-mail: zjuyjw@zju.edu.cn.*
- *Q. Li is with the Department of Computing, the Hong Kong Polytechnic University, Hong Kong, P. R. China. E-mail: csqli@comp.polyu.edu.hk.*

*Manuscript received XX XX, 2020; revised XX XX, 2020.*

TABLE 1
A Dataset of an Online Shopping System

(a) Buyer (B)

| Uid | User | Gender | Age | Address |
|---|---|---|---|---|
| $u_1$ | Who | M | 21 | $\perp$ |
| $u_2$ | Smile | $\perp$ | $\perp$ | Holywell Street |
| $u_3$ | *S** | F | 28 | Market Street |
| $u_4$ | Sky | $\perp$ | $\perp$ | $\perp$ |

(b) Transaction (T)

| Tid | Uid | Pid | Purchased time | Delivery time |
|---|---|---|---|---|
| $t_1$ | $u_1$ | $p_1$ | 2019-09-21 | 2019-09-29 |
| $t_2$ | $u_2$ | $p_1$ | 2019-10-22 | $\perp$ |
| $t_3$ | $\perp$ | $p_3$ | 2019-10-21 | 2019-11-01 |
| $t_4$ | $u_2$ | $\perp$ | 2019-09-22 | $\perp$ |

(c) Product (P)

| Pid | Size | Memory | Price |
|---|---|---|---|
| $p_1$ | 4.7 | 64 | \$5988 |
| $p_2$ | 4.7 | 256 | \$7288 |
| $p_3$ | 5.5 | 64 | \$6888 |
| $p_4$ | 5.5 | 256 | \$8188 |

the literature [13], [14], [15], [16], [17], [18], [19]. However, these existing pricing approaches are *not* suitable for pricing incomplete data, since they do not consider data completeness. To the best of our knowledge, this is the first attempt to investigate the problem of pricing incomplete data.

Consider an online shopping system data set, it has three relations Buyer (B), Transaction (T), and Product (P), as illustrated in Table 1. In particular, the buyer information shown in Table 1(a) is incomplete, due to privacy concerns from buyers. In addition, the missing transaction information, depicted in Table 1(b), results from the system failure or invalid data acquisition. The symbol "$\perp$" denotes the missing value (throughout the paper). Obviously, in this case, the assumption that every tuple has the *same* price is *unfair*. Intuitively, the prices of tuples $u_3$ and $u_4$ in Buyer should be distinguished, as $u_3$ owns more complete/observed attribute values than $u_4$.

Assume that a data analyst, Alice, is going to conduct an investigation for a new smartphone. For the sake of economy, Alice is willing to pay several queries over the dataset, instead of purchasing the whole data set with the three relations at a pretty price. To begin with, she posts a query $Q_1^\sigma$ (i.e., `SELECT * FROM T WHERE pid =` $p_1$) to get the transaction information about the product $p_1$ (with 4.7in size and 64G memory). Then, she gets the transaction information of $p_2$ (with 4.7in size and 256G memory), via posting the query $Q_2^\sigma$ (i.e., `SELECT * FROM T WHERE pid =` $p_2$). Alternatively, she would like to know the transaction information of $p_1$ and $p_2$ (with 4.7in size) directly, by posting the query $Q_3^\sigma$ (w.r.t. `SELECT * FROM T WHERE pid =` $p_1$ `OR pid =` $p_2$). In this scenario, how do we price the queries effectively and reasonably? What if it meets missing values?

Therefore, trading incomplete data is faced with a series of challenges, e.g., how reliable is the decision made over incomplete data, how does the missing information affect the final decision, and what exactly influences the probably unreliable query result. Hence, practical price functions are desirable to sell incomplete data, and we systematically study the pricing problem over incomplete data.

In this paper, we propose a *sophisticated pricing mechanism*, termed as iDBPricer, built on top of the *data contribution/usage*, *data completeness*, and *query quality*. First, iDBPricer assigns a selling price to a query by taking into account *data contribution* to the query (via leveraging the query *lineage*). The query lineage provides a way to measure the data contribution for queries in the field of database, and the lineage tuples can be treated as *factors of production* in microeconomics. In particular, the completeness degree of those lineage tuples is critical to measure the value of the query, and is considered into iDBPricer. Therefore, we define an effective usage and completeness-aware price function (*UCA price* for short) for selling incomplete data. Furthermore, iDBPricer takes into account the query quality (which is measured by the query result uncertainty). Thus, we present a novel quality, usage, and completeness-aware price function (*QUCA price* for short). In a nutshell, our key contributions are summarized as follows.

- We identify the pricing problem of queries on incomplete data, and we propose a sophisticated pricing mechanism, i.e., iDBPricer.
- We present two effective query price functions via considering the data *contribution*, data *completeness*, and query *quality*, and then, we develop efficient strategies and algorithms to derive the query prices.
- Extensive experiments using both real and benchmark datasets demonstrate the effectiveness and efficiency of our proposed pricing scheme iDBPricer, compared to the state-of-the-art pricing methods.

The rest of this paper is organized as follows. We present some preliminaries in Section 2. The two price functions are formalized in Section 3 and Section 4, respectively. Section 5 elaborates efficient algorithms for calculating the query price. Experimental results and our findings are reported in Section 6. Finally, we overview the related work in Section 7, and we conclude the paper with some directions for future work in Section 8.

## 2 Background

In this section, we introduce the query on incomplete data and the problem studied in this paper. Table 2 summarizes the symbols used frequently throughout the paper.

An incomplete dataset, denoted as $\mathbb{D}$, consists of a set of complete datasets $\{D_1, D_2, \cdots\}$ (called possible worlds). A possible world is a complete dataset by replacing each missing value in $\mathbb{D}$ with a constant, and the constant originates from the tuple values in $\mathbb{D}$. A query over an incomplete dataset retrieves the true answers (a.k.a. *certain answers*), as stated in Definition 1.

***Definition 1.*** **(Query over incomplete data)**. Given a query $Q$ with respect to an incomplete dataset $\mathbb{D}$, the query result set, denoted by $Q(\mathbb{D})$, is defined as the certain answers of $Q$ over $\mathbb{D}$, which is formalized by Eq. 1. Specifically, $Q(D)$ is the result set of $Q$ over the possible world $D$ of $\mathbb{D}$.

$$Q(\mathbb{D}) = \bigcap_{D \in \mathbb{D}} Q(D) \tag{1}$$

It is necessary to mention that, for a query $Q$, the objects in an incomplete dataset can be divided into three sets [20], [21]. In particular, one object set consists of objects (w.r.t. *true* answers) that are always answers to $Q$ over each possible

TABLE 2
Symbols and Description

| Notation | Description |
|---|---|
| $\mathbb{D}$ | an incomplete dataset |
| $Q$ | a query or an operator |
| $Q(\mathbb{D})$ | the result set of a query $Q$ over a dataset $\mathbb{D}$ |
| $t_i$ | a tuple belonging to a dataset $\mathbb{D}$ |
| $\rho(t_i)$ | the price of a tuple $t_i$ |
| $\eta_i$ | the complete rate of a tuple $t_i$ |
| $\Phi(Q, \mathbb{D})$ | the UCA price of a query $Q$ w.r.t. $\mathbb{D}$ |
| $\epsilon(t_i)$ | the quality of a tuple $t_i$ |
| $\kappa(Q, \mathbb{D})$ | the query quality function of $Q$ w.r.t. $\mathbb{D}$ |
| $\Phi_e(Q, \mathbb{D})$ | the QUCA price of a query $Q$ w.r.t. $\mathbb{D}$ |

world, i.e., $Q(\mathbb{D})$, corresponding to Definition 1. In contrast, the second object set is composed of objects (w.r.t. *unknown* answers) that are answers to $Q$ with respect to some, but not necessarily every, possible worlds. The unknown answers result from the data incompleteness. It is closely related to the query result uncertainty, which will be analyzed later. Except for the former two sets, the remaining objects are included in the third object set, which are certainly *false* answers (from the dataset) to the query $Q$.

In this paper, we mainly consider SPJ queries, viz., selection, projection, and natural join (without self-joins or predicates), denoted as $\sigma$, $\pi$, and $\bowtie$, respectively. As a result, the returned (certain) answers according to Definition 1 do not have any missing value over selected/projected attributes of selection and projection or keys of join queries. In addition, please note that we use the terms "query" and "operator" interchangeably throughout the paper.

***Example 1.*** Take the sample dataset in Table 1 as an example. For the selection query $Q_1^{\sigma}$ (defined in Section 1), it retrieves the tuples $t_1$ and $t_2$, according to the query definition. This is because, for any possible world, only $t_1$ and $t_2$ are always true query answers.

$Q_4^{\bowtie}$: `SELECT * FROM T, P WHERE T.pid=P.pid`
$Q_5^{\pi}$: `SELECT DISTINCT Size FROM P`

For the join query $Q_4^{\bowtie}$, there are three result tuples, i.e., $\{r_1, r_2, r_3\}$ (as plotted in Table 3), since they are query answers over all possible worlds. In contrast, as shown in Table 1(b), the $pid$ of $t_4$ in relation Transaction is missing. It is worth noting that, only when the pid of the tuple, i.e., $t_4$.pid, is equal to $p_i$ $(i = 1, \cdots, 4)$, $t_4$ is able to join with the tuples from the relation Product. In addition, the projection query $Q_5^{\pi}$ returns {'4.7', '5.5'} as the query result set, according to Definition 1.

In the pricing model of this paper, each tuple $t$ in a dataset $\mathbb{D}$ has a base price, defined by a price function $\rho(t)$. It is irrespective of the tuple's completeness. The price function $\rho$ generates a price per tuple. This is a common way to price relational data in real data markets [22]. The base price $\rho(t)$ is determined by the data price manager or the data owner, and prepared before the real data trade through the data cost estimation and market investigations. For example, according to the expectancy-value theory[1] in social psychology, the base price $\rho$ of a tuple $t$ can be estimated by $t$'s valuation from the seller and $t$'s obtained cost, i.e., $\rho(t) = \mathcal{V}(t) \cdot \mathcal{C}(t)$. In particular, the harvesting cost $\mathcal{C}$ (resp. the tuple valuation $\mathcal{V}$) follows some real-life distributions. Consequently, different base tuples take respective prices.

1. https://en.wikipedia.org/wiki/Expectancy-value_theory

TABLE 3
The Result Tuples of a Query $Q_4^{\bowtie}$

| Rid | Uid | Pid | Size | Memory | Price | Purchased Time | Delivery Time |
|---|---|---|---|---|---|---|---|
| $r_1$ | $u_1$ | $p_1$ | 4.7 | 64 | \$5988 | 2019-09-22 | 2019-12-22 |
| $r_2$ | $u_2$ | $p_1$ | 4.7 | 64 | \$5988 | 2019-10-21 | $\perp$ |
| $r_3$ | $\perp$ | $p_3$ | 5.5 | 64 | \$6888 | 2019-10-21 | 2019-12-22 |

Therefore, given a query $Q$ with respect to an incomplete dataset $\mathbb{D}$, the pricing problem for queries over incomplete data is to assign a price $\Phi$ for a query $Q$ over an incomplete data set $\mathbb{D}$, such that the price is reasonable and practical. It assumes that, all tuples have the same base price throughout the paper. Our proposed solution could generalize to the cases with non-uniform base prices.

## 3 The UC-Aware Price Function

In this section, we formalize an effective data usage and completeness-aware (UC-aware, i.e., UCA) price function.

First, we evaluate the data contribution/usage via the concept of *data lineage* from the perspective of *data provenance* in the database field. Specifically, each tuple $t$ occurs in the output of a query with a set of tuples presented in the input, called the lineage of $t$ [23]. Intuitively, the lineage of $t$ is meant to collect all of the input data that "contribute to" $t$ or help to "produce" $t$, as stated in Definition 2.

We would like to mention that, this data lineage-based solution to measure data contribution is different from the Shapley value based-pricing methods for machine learning models [17], which are *not* able to directly support the pricing problem on incomplete data, due to the different characteristics of the studied problems.

***Definition 2.*** **(Tuple's lineage set)**. Given a dataset $\mathbb{D}$ with tables $T_1, \cdots, T_m$, and a query $Q$ (i.e., $\sigma$, or $\pi$, or $\bowtie$). Let $Q(\mathbb{D}) = Q(T_1, \cdots, T_m)$ be the result set of the query $Q$ over tables $T_1, \cdots, T_m$. For a tuple $t \in Q(\mathbb{D})$, one $t$'s lineage set w.r.t. $Q$ in $T_1, \cdots, T_m$, denoted as $L(t \in Q(\mathbb{D}), \mathbb{D})$ ($L(t, \mathbb{D})$ for short), is defined by Eq. 2.

$$L(t, \mathbb{D}) = \bigcup_{i=1}^{m} T_i^* \tag{2}$$

$$Q^{-1}_{\langle T_1, \cdots, T_m \rangle}(t) = \langle T_1^*, \cdots, T_m^* \rangle \tag{3}$$

Eq. 3 is the vector form of a lineage set of $t$, with each element $T_i^*$ having tuples from the table $T_i$. For $i = 1, \cdots, m$, $Q^{-1}_{T_i}(t) = T_i^*$ is $t$'s lineage in $T_i$, and each tuple in $T_i^*$ does contribute to the result tuple $t$. Formally, $T_1^*, \cdots, T_m^*$ are subsets of $T_1, \cdots, T_m$ satisfying
(a) $Q(T_1^*, \cdots, T_m^*) = \{t\}$;
(b) $\forall T_i^*, \forall T^{'} \subseteq T_i^*, Q(T_1^*, \cdots, T^{'}, \cdots, T_m^*) = \varnothing$.

In fact, the condition (a) constrains that, the lineage tuple sets (i.e., $T_i^*$'s) derive exactly $t$, and the condition (b) indicates that, each tuple in the lineage indeed contributes something to $t$.

***Example 2.*** For the sample dataset (denoted by $\mathbb{D}_1$) in Table 1, in terms of the projection query $Q_5^{\pi}$, its result tuples are {'4.7', '5.5'}, as shown in Example 1. According to Definition 2, the result tuple '4.7' has two lineage

sets, i.e., $L_1$('4.7', $\mathbb{D}_1$) = $\{p_1\}$ and $L_2$('4.7', $\mathbb{D}_1$) = $\{p_2\}$, both of which can produce the tuple '4.7'. Similarly, for the result tuple '5.5', it has two lineage sets, namely, $L_1$('5.5', $\mathbb{D}_1$) = $\{p_3\}$ and $L_2$('5.5', $\mathbb{D}_1$) = $\{p_4\}$. Consequently, there are four lineage sets for the query $Q_5^\pi$: namely, $M_1(Q_5^\pi, \mathbb{D}_1) = \{p_1, p_3\}$, $M_2(Q_5^\pi, \mathbb{D}_1) = \{p_1, p_4\}$, $M_3(Q_5^\pi, \mathbb{D}_1) = \{p_2, p_3\}$, and $M_4(Q_5^\pi, \mathbb{D}_1) = \{p_2, p_4\}$. Thus, the lineage set collection of $Q_5^\pi$, i.e., $\mathcal{M}(Q_5^\pi, \mathbb{D}_1) = \{\{p_1, p_3\}, \{p_1, p_4\}, \{p_2, p_3\}, \{p_2, p_4\}\}$.

Throughout this paper, we use $\mathcal{L}(t, \mathbb{D}) = \{L(t, \mathbb{D})\}$ to collect all the lineage sets for the result tuple $t$ (to adapt the case of $\pi$). Then, one lineage set of a query result set $Q(\mathbb{D})$, denoted as $M(Q, \mathbb{D})$, is the union set of one lineage set $L(t, \mathbb{D})$ from each result tuple $t \in Q(\mathbb{D})$, namely, $M(Q, \mathbb{D}) = \bigcup_{t \in Q(\mathbb{D})} L(t, \mathbb{D})$. It indicates that, the data usage of the query $Q$ over $\mathbb{D}$ is evaluated by $M(Q, \mathbb{D})$.

It is worth noting that, for selection and join operators (i.e., $\sigma$ and $\bowtie$), each of result tuples only has one lineage set, i.e., $|\mathcal{L}(t, \mathbb{D})| = 1$. For example, given a selection operator, for the tuples that do not satisfy the selection condition, they make no contribution to any result tuple. Hence, these tuples will not appear in any result tuple's lineage. As a result, the lineage set of operators $\sigma$ and $\bowtie$ is unique. In contrast, for a projection operator $\pi$, there are probably several lineage sets for a result tuple $t \in Q(\mathbb{D})$, since two or more tuples are likely to be projected into one result tuple $t$. Thus, for a projection operator (i.e., $\pi$), there is one or more lineage set(s) for each result tuple, i.e., $|\mathcal{L}(t, \mathbb{D})| \geq 1$.

As a consequence, Definition 3 formulates the data usage and completeness-aware query price function for queries on incomplete data, by taking into account both the data contribution and data completeness.

***Definition 3.*** **(The UCA price function)**. Given an incomplete dataset $\mathbb{D}$, the data usage and completeness-aware price (i.e., UCA price) of a query $Q$, denoted by $\Phi(Q, \mathbb{D})$, is defined as the accumulated weighted prices of the lineage tuples of $Q$.

$$\Phi(Q, \mathbb{D}) = \sum_{t_i \in M(Q, \mathbb{D})} \eta_i \cdot \rho(t_i) \tag{4}$$

In particular, $\eta_i$ represents the complete rate of the tuple $t_i$, i.e., the ratio of the number of $t_i$'s observed values to the total number of $t_i$'s attributes $d$, $\rho(t_i)$ is the base price of $t_i$, and $M(Q, \mathbb{D})$ denotes a lineage set of the query $Q$ over the dataset $\mathbb{D}$ (w.r.t. the data usage of $Q$).

The basic idea of the UCA price is to discount the price depending on tuples' completeness degree over the lineage tuples (contributed to query answers). The price function is reasonable and practical, since it utilizes the data lineage to measure the data contribution. Meanwhile, the completeness degree of lineage tuples is critical to measure the value of the query. The price of $\eta_i \cdot \rho(t_i)$ is a sort of discount price based on the tuple's completeness degree. As an exception, if the result set is empty, the query price could be determined by the seller, e.g., referring to the current market situation. For the projection operator, it assigns it the cheapest price of all possible lineage sets, using Eq. 4. One can also compute other combined prices (e.g., the highest/average price) of lineage tuples for it. Moreover, the higher the tuple completeness, the higher the UCA price.

The arbitrage-free property widely explored in relevant studies [13], [16], [19] is a typical and important property for price functions. This property is necessary for data pricing to *avoid arbitrage*. An arbitrage-free price function means that, the data buyer cannot purchase a query by buying a group of other queries at a lower price, and the price of a given query is unique and same. In particular, the arbitrage-free property is defined based on a notion of *query determinacy* [14]. Informally, a set of data views/queries $\mathbf{V}$ determines some query $Q$, if one can compute answers of $Q$ only from answers of views without having access to the underlying dataset. Put it differently, $\mathbf{V}$ determines $Q$ if $\mathbf{V}$ provides enough information to uniquely determine answers to $Q$. Specifically, given a dataset $\mathbb{D}$, $\mathbf{V}$ determines $Q$ (denoted as $\mathbb{D} \vdash \mathbf{V} \twoheadrightarrow Q$) if one can answer $Q$ from answers of $\mathbf{V}$ by applying a function $f$ such that $Q(\mathbb{D}) = f(\mathbf{V}(\mathbb{D}))$. The impact on pricing is that if the user needs to answer the query $Q$, he/she has the option of querying $\mathbf{V}$ and then applying $f$. If $\mathbf{V}$ determines $Q$, a potential buyer interested in purchasing $Q$ can buy $\mathbf{V}$ instead, and derive from its answers to $Q$: arbitrage *occurs* when the price of $\mathbf{V}$ is lower than that of $Q$.

***Property 1.*** The query price function $\Phi$ defined by Eq. 4 is *arbitrage-free*. It is confirmed that, for a set of views $\mathbf{V}$ and a query $Q$ over a dataset $\mathbb{D}$, if $\mathbf{V}$ determines $Q$, the price of $\mathbf{V}$ is not lower than that of $Q$, namely, $\Phi(Q, \mathbb{D}) \leq \Phi(\mathbf{V}, \mathbb{D})$, if $\mathbb{D} \vdash \mathbf{V} \twoheadrightarrow Q$.

According to the determinacy definition above, there exists a function $f$ such that $Q(\mathbb{D}) = f(\mathbf{V}(\mathbb{D}))$. Hence, we can know that, the lineage of $\mathbf{V}$ covers the lineage set of $Q$. That is to say, for any lineage set $M$ of $Q$, there exists certainly a union set of the lineage set of each view from $\mathbf{V}$ that containing $M$. Therefore, we can derive $\Phi(Q, \mathbb{D}) \leq \Phi(\mathbf{V}, \mathbb{D})$, based on Definition 3.

***Example 3.*** The UCA price of $Q_4^\bowtie$ is calculated as follows. For the lineage set $\{t_1, t_2, t_3, p_1, p_3\}$ of the query $Q_4^\bowtie$, the complete rate of every lineage tuple is firstly derived. Specifically, the complete rate of $t_1$, i.e., $\eta(t_1)$, is 1, as its attribute values are all completed. The complete rate of $t_2$, i.e., $\eta(t_2)$, is $\frac{3}{4}$, since it misses the value on the attribute "Delivery time". Similarly, we have $\eta(t_3) = \frac{3}{4}$ and $\eta(p_1) = \eta(p_3) = 1$. Therefore, if we assume that the base price of each tuple $\rho = 10$, the UCA price of $Q_4^\bowtie$, i.e., $\Phi(Q_4^\bowtie, \mathbb{D}_1) = 10 \times (1 + \frac{3}{4} + \frac{3}{4} + 1 + 1) = 45$, according to Definition 3.

## 4 THE QUC-AWARE PRICE FUNCTION

In this section, we present an improved price function, which considers *query quality*, data contribution, and data completeness. Then, we analyze the extendibility of our pricing mechanism for adapting to various scenarios.

One can observe that, the UCA price in Definition 3 involves the lineage tuples and the missing degree of them. It estimates the query from the aspect of the missing rate in lineage tuples, which reflects the degree of values missing in lineage tuples. However, it neglects the query result uncertainty, such as the degree of true tuples absent from answers. In light of this, we attempt to present an enhanced price function, namely, the quality, usage, and completeness-aware (QUC-aware, i.e., QUCA) price.

First, for an incomplete dataset $\mathbb{D}$, we define a *critical attribute* set, denoted by $A_C^Q$, as the set of constraint attributes for $Q$ being $\sigma$, or projected attributes for $Q$ being $\pi$, or *foreign keys* for $Q$ being $\bowtie$. In other words, the critical attribute set $A_C^Q$ is defined to exactly cover all the attributes on which any missingness might incur "unknown" query results.

Hence, we call tuples with missing values on critical attributes $A_C^Q$ as *uncertain* lineage tuples. The *uncertainty* of the uncertain lineage tuples is that, their contribution to query results are *uncertain*, making the query produce unknown answers. According to Definition 1 and Definition 2, one can conclude that, the lineage of the query result set $Q(\mathbb{D})$ *excludes* the tuples with one or more missing values on attributes $A_C^Q$. While the uncertain lineage tuples have the potential to be (or contribute to) *unknown answers* of the query $Q$. In contrast, *certain* lineage tuples correspond to those defined in Definition 2.

Second, for a tuple $t_i \in \mathbb{D}$, we use a parameter $x_i$ to count the number of the missing values in critical attributes $A_C^Q$. Namely, $x_i = |\{j \in A_C^Q \mid t_i.[j] \text{ is missing}\}|$. Therefore, the quality of tuple $t_i$, denoted by $\epsilon(t_i)$, is defined by Eq. 5.

$$\epsilon(t_i) = a^{x_i} \tag{5}$$

In particular, $a$ ($0 < a < 1$) indicates the sensitivity degree of the users to the quality, which can be specified by the data pricing manager or the data owner.

The parameter $a$ can be dynamically tuned for a data consumer based on his/her history purchases. It helps to carry out the market segmentation of selling incomplete data. To some extent, $a$ measures the changing degree of tuple quality w.r.t. $x_i$. The smaller the value of $a$, the quickly the quality value changes (namely, the more sensitive the quality is to $x_i$). Obviously, the tuple quality $\epsilon$ emphasizes the critical attributes of a specific query, instead of equally treating each attribute. Here, the exponential function $a^{x_i}$ is chosen because of its perfect property matching the tuple quality metric. Since we aim at pricing incomplete data, we mainly pay attentions on the query quality caused by data incompleteness, and other forms of the tuple quality functions can be also applied to different real-life scenarios.

Let $\langle \mathcal{T}_1^*, \cdots, \mathcal{T}_m^* \rangle$ be the vector form of one lineage set w.r.t. the query $Q$ over tables $T_1, \cdots, T_m$ from $\mathbb{D}$ (w.r.t. true answers). Namely, $\mathcal{T}_i^* = \bigcup_{t \in Q(\mathbb{D})} Q_{T_i}^{-1}(t)$. Let $\langle T_1^a, \cdots, T_m^a \rangle$ denote the vector form of uncertain lineage tuples that miss some values on critical attributes $A_C^Q$. As a result, a quality function to measure the query result uncertainty is formalized in Definition 4.

***Definition 4.*** **(The query quality function)**. Given an incomplete dataset $\mathbb{D}$ and a query $Q$. The quality of the query $Q$, denoted by $\kappa(Q, \mathbb{D})$, is defined in Eq. 6.

$$\kappa(Q, \mathbb{D}) = \frac{\sum_{i=1}^{m} (\sum_{t \in \mathcal{T}_i^*} \epsilon(t) + \sum_{t \in T_i^a} \epsilon(t))}{\sum_{i=1}^{m} (|\mathcal{T}_i^*| + |T_i^a|)} \tag{6}$$

The newly presented quality function $\kappa$ takes into account the specific operator, which is different from the simple quality of counting the missing rate on the entire dataset [24], [25]. It provides a new way to measure the query quality over incomplete data. Specifically, the quality function $\kappa$ is defined as the average quality of (both *certain* and *uncertain*) lineage tuples in terms of a specific operator $Q$, where the tuple quality (denoted by $\epsilon$) is based on an exponential function and treats missing *critical* attributes w.r.t. $Q$ *more importantly*.

Moreover, if the set of uncertain lineage tuples is empty, the query quality $\kappa(Q, \mathbb{D})$ is equal to one, meaning that there is no *unknown* query result (that are possibly, but not certainly, query answers) for this operator. Hence, the more the unknown query answers, the more the tuples with lower quality, and thus, the lower the query quality $\kappa(Q, \mathbb{D})$. Depending on the property of the quality function $\epsilon(t)$, one can conclude that, the value of $\kappa(Q, \mathbb{D})$ is in the range (0, 1].

***Example 4.*** In terms of the join query $Q_4^{\bowtie}$ (as defined in Example 1), its result tuples are shown in Table 3. It is easy to get that, the lineage set of $Q_4^{\bowtie}$, denoted by $M_1(Q_4^{\bowtie}, \mathbb{D})$, is the set of $\{t_1, t_2, t_3, p_1, p_3\}$. Note that, the tuples in $M_1(Q_4^{\bowtie}, \mathbb{D})$ come from the relations Transaction and Product, as illustrated in Table 1. In addition, the uncertain lineage is the set $\{t_4\}$, because it misses the value $pid$ (as depicted in Table 1(b)). Assume that $a = \frac{1}{2}$. According to the tuple quality defined in Eq. 5, we have $\epsilon(t_1) = a^{x_1} = 1$, since there is no missing information on the foreign key (i.e., $pid$) of $t_1$ in the relation Transaction (i.e., $x_1 = 0$). Similarly, we have $\epsilon(t_2) = \epsilon(t_3) = \epsilon(p_1) = \epsilon(p_3) = 1$ and $\epsilon(t_4) = \frac{1}{2}$. Therefore, according to Eq. 6, the query quality $\kappa(Q_4^{\bowtie}, \mathbb{D}_1) = \frac{1+1+1+1+1+\frac{1}{2}}{5+1} = \frac{11}{12}$.

The UCA price function in Definition 3 to some extent reveals the output information (e.g., the scale of result set, and thereby the lineage set). Hence, it might leak some result information to the data buyers. As a result, in addition to taking the query quality into consideration, we try to make the query price insensitive to the result set scale. Thus, the improved price function is presented in Definition 5.

***Definition 5.*** **(The QUCA price function)**. For a query $Q$ over an incomplete dataset $\mathbb{D}$, the quality, usage, and completeness-aware query price (QUCA price for short), denoted by $\Phi_e(Q, \mathbb{D})$, is written in Eq. 7, where $n$ is the size of result tuples, i.e., $n = |Q(\mathbb{D})|$, and $\Delta$ is a price coefficient used to control the price scale by users.

$$\Phi_e(Q, \mathbb{D}) = \frac{1}{n} \cdot \Delta \cdot \kappa(Q, \mathbb{D}) \cdot \Phi(Q, \mathbb{D}) \tag{7}$$

With considering the size of results tuples, the QUCA price not only considers the missing state and quality of lineage tuples, but also prevents it from revealing some information of the result set and lineage set size. Moreover, the price coefficient $\Delta$ provides an entrance to tune the amount of price flexibly for adapting the data market environment. When the data owner has other requirements for data pricing, he/she can convert the parameter $\Delta$ to be a function. The parameter $\Delta$ allows to incorporate more factors to enhance the QUCA price function. In addition, it is simple to use $\Delta$ to control the whole price range, keeping the data price in a reasonable range.

Thus, QUCA is more practical than UCA. While the arbitrage-free property of UCA price function is non-transitive to the QUCA price function, due to the consideration of normalized query quality in the QUCA price.

***Example 5.*** In terms of the join query $Q_4^{\bowtie}$, the query result size is equal to 3, as plotted in Table 3. The UCA price

is equal to 45, referring to Example 3. The query quality is $\frac{11}{12}$, as calculated in Example 4. Assume that the price coefficient $\Delta$ is one. the QUCA query price of $Q_4^{\bowtie}$, i.e., $\Phi_e(Q_4^{\bowtie}, \mathbb{D}_1) = \frac{1}{3} \cdot \frac{11}{12} \cdot 45 = \frac{55}{4}$.

**The history-aware price case.** When Alice has purchased $Q_1^{\sigma}$ and $Q_2^{\sigma}$ (described in Section 1), she decides to explore all information in Transaction via a *new* query. At that time, she probably pays for the fraction of the new query that she has paid in purchasing $Q_1^{\sigma}$ or $Q_2^{\sigma}$. When considering the *historical* queries (instead of solely considering the current query), the presented price functions above are useless. One has to resort to the *history-aware* price function [15].

Intuitively, one should ensure that, Alice does not pay for the fraction of the new query that she has paid in purchasing $Q_1^{\sigma}$ or $Q_2^{\sigma}$ *any more*. Thus, the pricing mechanism should prevent the buyer from overpaying when he/she has already acquired some information from previous/historical queries. For instance, given an incomplete dataset $\mathbb{D}$ and a new query $Q$ from a customer, if the customer has purchased a series of queries $Q_1, Q_2, \cdots, Q_x$, it indicates that, the customer has bought the corresponding lineage tuples of these queries. Namely, the pricing mechanism could claim that, if there's no new update in any information of $t$, he/she has rights to re-use the lineage tuple $t$ of these queries.

As a result, in the history-aware case, the two price functions, i.e., UCA price and QUCA price, could be derived via free charge for the *active* lineage tuples of historical queries (that the customer has paid). Let $T$ collects all the active lineage tuples (that he/she has rights to re-use), the history-aware QUCA price of the query $Q$, denoted by $\Phi_e^h(Q, \mathbb{D}, T)$, can be defined as $\frac{1}{n} \cdot \Delta \cdot \kappa(Q, \mathbb{D}) \cdot \sum_{t_i \in M \setminus T} \eta_i \cdot \rho(t_i)$. Hence, it avoids repeated charge for historical queries.

**Discussion.** Our price functions not only consider the intrinsic properties of datasets and economic factors in data markets, but also offer a generic pricing framework to adapt to various situations. Specifically, our price functions do not limit the specific forms of the base tuple price function $\rho(t)$ and the tuple quality function $\epsilon(t)$ (defined in Eq. 5). Namely, they are *free* for our price functions, and the forms of them do not affect the applicability of our pricing mechanism and solutions. Moreover, our proposed pricing mechanism can support the considerations on different database levels (e.g., the attribute, tuple, and table levels). It allows data markets to incorporate their own pricing preferences into it, so as to generate a practical price function that satisfies the pricing demands in various real-life scenarios.

For example, the base price function and the tuple quality function can incorporate the importance degree of each table/attribute (as well as the tuple semantic). If one assigns a weight on each attribute (or table) to indicate the importance, the base price of a tuple can be defined by accumulating the attribute (or table) weights of the tuple. In addition, when one adds a weight on each attribute, the parameter $x_i$ in the tuple quality function can be defined as the sum of weights on missing critical attributes.

## 5 Price Computation

In this section, we first explore the lineage set derivation methods. Then, we propose efficient query price computation algorithms. For clarity, we illustrate the solution of deriving the QUCA price. They are applicable to the UCA price and the history-aware price.

**Algorithm 1:** Baseline

**Input**: lineage sets of each result tuple from $Q(\mathbb{D})$, a price coefficient $\Delta$, a query result cardinality $n$
**Output**: the QUCA price $\Phi_e(Q, \mathbb{D})$

1: $\Phi(Q, \mathbb{D}) \longleftarrow +\infty$ `// Initialize the price`
2: get every query lineage set $M$ via selecting exactly one lineage set from each $\mathcal{L}(t_i)$ with $t_i \in Q(\mathbb{D})$
3: **foreach** query lineage set $M$ **do**
4: $\quad \Phi_0 \longleftarrow \sum_{t_j \in M} \eta_j \cdot \rho(t_j)$
5: $\quad \Phi(Q, \mathbb{D}) \longleftarrow \min(\Phi(Q, \mathbb{D}), \Phi_0)$
6: compute $\kappa(Q, \mathbb{D})$ based on Eq. 6
7: $\Phi_e(Q, \mathbb{D}) \longleftarrow \frac{\Delta \cdot \kappa(Q, \mathbb{D})}{n} \cdot \Phi(Q, \mathbb{D})$
8: **return** $\Phi_e(Q, \mathbb{D})$

### 5.1 Deriving the Lineage Sets

We utilize two strategies, i.e., Active and Lazy, to seek lineage sets. They are the most appropriate methods for finding lineage tuples [26]. In particular, the main idea of Active is to track the procedure of performing the query $Q$, and record lineage sets simultaneously. For the selection operator, it records directly the lineage set $L(t_i, \mathbb{D})$ of each result tuple $t_i$ as the set $\{t_i\}$. For the projection operator, no matter whether the query $Q$ projects $t_i$ into a new result tuple or not, Active updates the collection $\mathcal{L}$ of lineage sets for each tuple $t_i$ by inserting the set $\{t_i\}$ into $\mathcal{L}$. If the join operator performs successfully for two tuples $v_i$ and $t_j$, the tuples $v_i$ and $t_j$ are both inserted as one lineage set of the joined tuple via $v_i$ and $t_j$.

Alternatively, Lazy derives lineage sets from a totally different angle. In Lazy, the task of deriving minimal lineage sets is translated into a kind of selection query, which takes the query result set $Q(\mathbb{D})$ as an input. It seems like some sort of reverse engineering. Specifically, for each result tuple $t_i \in Q(\mathbb{D})$, Lazy judges whether the tuple from $\mathbb{D}$ could (partly) produce the result tuple $t_i$ via the query $Q$. If the answer is yes, the lineage set collection $\mathcal{L}(t_i, \mathbb{D})$ is updated by adding this tuple. In this way, all the lineage set collections $\mathcal{L}$s of result tuples in $Q(\mathbb{D})$ are obtained.

Since Active performs along with the operator processing, and it does not incur any extra overhead to derive the lineage set. Thus, Active enjoys the lower complexity. In contrast, Lazy obtains the lineage set after getting the result set of the operator, which makes it less efficient. Among SPJ operators, the join operator has the highest complexity. Let $n$ be the number of result tuples, and $d_1$ and $d_2$ be cardinalities of the two relations in join, respectively. The complexity of Active is $O(d_1 \cdot d_2)$, and that of Lazy is $O(n \cdot d_1 \cdot d_2)$.

### 5.2 Computing the Prices

**Baseline method.** First, leveraging lineage sets derived from Active (or Lazy) strategy, we compute the query price based on the price of the query lineage set, according to Definition 3. It is a straightforward method (called Baseline).

Algorithm 1 depicts the price computation procedure. It takes derived lineage sets of each result tuple from $Q(\mathbb{D})$

**Algorithm 2:** AD&C

**Input**: lineage sets of each result tuple from $Q(\mathbb{D})$, a price coefficient $\Delta$, a query result cardinality $n$
**Output**: the QUCA price $\Phi_e(Q, \mathbb{D})$

1: $\Phi(Q, \mathbb{D}) \longleftarrow +\infty$ `// Initialize the price`
2: **foreach** result tuple $t_i \in Q(\mathbb{D})$ **do**
3: get the lineage set $L_i^\star$ with the lowest price from $\mathcal{L}(t_i)$
4: $M \longleftarrow \bigcup_{t_i \in Q(\mathbb{D})} L_i^\star$
5: $\Phi(Q, \mathbb{D}) \longleftarrow \sum_{t_j \in M} \eta_j \cdot \rho(t_j)$
6: compute $\kappa(Q, \mathbb{D})$ based on Eq. 6
7: $\Phi_e(Q, \mathbb{D}) \longleftarrow \frac{\Delta \cdot \kappa(Q, \mathbb{D})}{n} \cdot \Phi(Q, \mathbb{D})$
8: **return** $\Phi_e(Q, \mathbb{D})$

as inputs, and outputs the query QUCA price $\Phi(Q, \mathbb{D})$. Initially, Baseline sets the price as infinity (line 1). Then, for every query lineage set $M$ (composed of exactly one lineage set from $\mathcal{L}$ for each result tuple), Baseline computes the price $\Phi(Q, \mathbb{D})$ based on Definition 3 (lines 2-5). Next, it derives the query result quality based on Eq. 6, and obtains the enhanced QUCA price $\Phi_e(Q, \mathbb{D})$ (lines 6-7). Finally, the algorithm terminates after returning $\Phi_e(Q, \mathbb{D})$ (line 8).

**AD&C algorithm**. We propose an adaptive D&C (AD&C for short) algorithm, which employs the *divide and conquer* strategy to calculate the price efficiently. The pseudo-code of AD&C is presented in Algorithm 2. It first chooses the lineage set, denoted as $L_i^\star$, with the lowest price for each result tuple $t_i$ (lines 2-3). Then, it gets the query price to be the QUCA price of the union set for all the chosen lineage sets $L_i^\star$s of result tuples (using Eq. 4) (lines 4-8).

Let $n$ be the number of result tuples, and $m$ be the average number of lineage sets for every result set. Algorithm 1 (i.e., Baseline) needs $O(m^n \cdot s)$ time to compute the price, as the number of query lineage sets is at most $m^n$ (for the operator projection $\pi$). In contrast, AD&C calculates prices at most $(m \cdot n + 1)$ times. In addition, Baseline is able to get the price accurately, while it cannot be done by AD&C. The approximate ratio AD&C is equal to $n$, i.e., the number of result tuples stored in $Q(\mathbb{D})$ [26]. The approximate ratio is the ratio of the approximate price (derived by AD&C) to the exact price. Note that, both Baseline and AD&C algorithms can be directly applied to calculate the history-aware price.

## 6 Experimental Evaluation

In this section, we present a comprehensive experimental evaluation on the pricing mechanism iDBPricer. All algorithms were implemented in C++, and all experiments were conducted on an Intel Core i7 Duo 3.60GHz PC with 24GB RAM, running Microsoft Windows 7 Professional Edition.

In our experiments, we use two public real-life datasets *world* [27], *DBLP* [28], and two benchmark datasets *TPC-H* [29], *SSB* [30], as described in Table 4. Specifically, *world* has three relations: Country, City, and CountryLanguage. *DBLP* is a co-authorship network. Two authors in DBLP are connected if they have a publication together. *TPC-H* is a performance metric benchmark for systems operating at a scale. *SSB* is a benchmark for measuring the performance of data warehouse style workloads. The database size of the benchmark datasets *TPC-H* and *SSB* is publicly measured by the scale factor, i.e., SF. For instance, SF = 1 on *TPC-H* means that, the corresponding dataset approximately occupies 1GB memory, and the rows of tables in *TPC-H* increase in proportion to SF. Following existing studies [31], [7], [10], we simulate incomplete datasets via randomly discarding some values at a certain missing rate.

TABLE 4
Used Datasets in the Experiments

| Dataset | # Relations | # Tuples | # Attributes |
|---|---|---|---|
| world | 3 | 5,302 | 24 |
| DBLP | 1 | 1,049,866 | 2 |
| TPC-H | 8 | SF = 1 | 61 |
| SSB | 5 | SF = 1 | 58 |

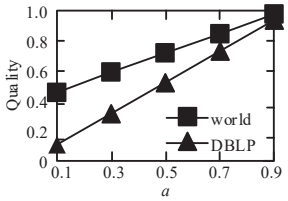


Fig. 2. Query quality vs. $a$

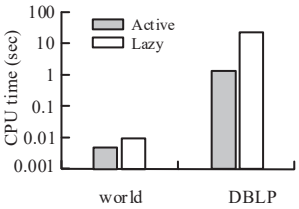


Fig. 3. Strategy study

In the experiments, following the related work [16], we use 40 queries over four datasets to conduct performance evaluation. All of the queries are described in Table 5, including 16 queries (w.r.t. $Q_1^w, \cdots, Q_{16}^w$) on *world*, 6 queries (w.r.t. $Q_1^d, \cdots, Q_6^d$) on *DBLP*, 10 queries (w.r.t. $Q_1^t, \cdots, Q_{10}^t$) over *TPC-H*, and 8 queries (w.r.t. $Q_1^s, \cdots, Q_8^s$) over *SSB*, respectively. These queries are randomly generated, and the number of queries makes no difference to the experimental results, since we report the average result value of the queries over the corresponding dataset. In the experiments, we study the effect of several factors on our algorithm performance, e.g., the missing rate and the data scale of the benchmark datasets *SF* (which are set by default to 0.2 and 1, respectively). In particular, the missing rate is the ratio of the number of missing values to the number of all values in the data set. The base price $\rho$ of each tuple is set to 1 for each set of experiments. The price coefficient $\Delta$ is set as 100 and 5,000 on the real datasets and benchmark datasets, respectively. The derived query prices and the execution time are the main metrics in the experiments. In each set of experiments, unless mentioned otherwise, only one parameter varies and others are set to default values.

We evaluate our proposed UCA and QUCA prices, compared with two baseline price functions, i.e., the *uniform* price (UP for short) and the *uniform completeness-based* price (UCP for short). Note that, existing pricing methods for *complete* data are *incomparable* to our pricing methods for incomplete data. In particular, UP assigns a fixed price amount (e.g., 1 in the experiments) to each query answer. Its idea comes from [19], which is common to treat all data in a unified manner in practice [32]. In other words, the uniform price UP is equal to the accumulated prices of all query answers. Furthermore, UCP is defined as the multiplication of UP and the dataset complete rate (i.e., one minus the missing rate), which is more reasonable as it considers the data completeness. We generalize existing methods [26] to implement Active and Lazy strategies, as they are the latest strategies to derive the query lineage set.

TABLE 5
Evaluated Queries in the Experiments

| ID | Query |
|---|---|
| $Q_1^w$ | SELECT Name FROM Country WHERE Continent = 'Asia' |
| $Q_2^w$ | SELECT Continent FROM Country |
| $Q_3^w$ | SELECT Language FROM CountryLanguage |
| $Q_4^w$ | SELECT Name FROM Country WHERE Region = 'Caribbean' |
| $Q_5^w$ | SELECT Name FROM Country WHERE Population between 10000000 and 20000000 |
| $Q_6^w$ | SELECT * FROM Country |
| $Q_7^w$ | SELECT * FROM Country WHERE Continent = 'Europe', Population $>$ 3000000 |
| $Q_8^w$ | SELECT * FROM Country WHERE SurfaceArea $>$ 1000000 |
| $Q_9^w$ | SELECT * FROM City T, Country C WHERE T.Code = C.Code, T.Name = 'NewCastle' |
| $Q_{10}^w$ | SELECT GovernmentForm FROM Country |
| $Q_{11}^w$ | SELECT * FROM City WHERE Population $>$ 1000000, CountryCode = 'USA' |
| $Q_{12}^w$ | SELECT * FROM CountryLanguage WHERE IsOfficial = 'T' |
| $Q_{13}^w$ | SELECT * FROM CountryLanguage WHERE CountryCode = 'USA' |
| $Q_{14}^w$ | SELECT * FROM City WHERE CountryCode = 'GRC' |
| $Q_{15}^w$ | SELECT * FROM CountryLanguage L, Country C WHERE L.Code = C.Code, L.Language = 'English', L.Percentage $>$ 50 |
| $Q_{16}^w$ | SELECT * FROM CountryLanguage L, Country C WHERE L.Code = C.Code, L.Language = 'Spanish' |
| $Q_1^d$ | SELECT * FROM DBLP WHERE FromNodeId $<$ 150 |
| $Q_2^d$ | SELECT ToNodeId FROM DBLP WHERE FromNodeId = 148255 |
| $Q_3^d$ | SELECT * FROM DBLP WHERE ToNodeId = 38868 |
| $Q_4^d$ | SELECT * FROM DBLP WHERE FromNodeId $<$ 10000, ToNodeId = 38868 |
| $Q_5^d$ | SELECT * FROM DBLP WHERE FromNodeId $<$ 10000, ToNodeId = 38828 |
| $Q_6^d$ | SELECT * FROM DBLP WHERE ToNodeId = 425000 |
| $Q_1^t$ | SELECT RegionKey FROM Nation N |
| $Q_2^t$ | SELECT * FROM Customer C, Nation N WHERE C.RegionKey = N.RegionKey |
| $Q_3^t$ | SELECT * FROM Supplier S, PartSupp PS WHERE S.SuppKey = PS.SuppKey, 500 $<$ SuppKey $<$ 1000 |
| $Q_4^t$ | SELECT * FROM LineItem L WHERE Quantity = 10, Discount $>$ 0.08 |
| $Q_5^t$ | SELECT * FROM Orders O WHERE TotalPrice $>$ 300000, OrderPriority = '1-Urgent' |
| $Q_6^t$ | SELECT * FROM Part P WHERE Brand = 'Brand#13' |
| $Q_7^t$ | SELECT * FROM Customer C WHERE CustKey $>$ 145000, NationKey = 13 |
| $Q_8^t$ | SELECT * FROM Orders O, Customer C WHERE O.CustKey = C.CustKey, CustKey $<$ 5000, TotalPrice $>$ 200000 |
| $Q_9^t$ | SELECT NationKey FROM Supplier S |
| $Q_{10}^t$ | SELECT * FROM LineItem L WHERE 8000 $<$ PartKey $<$ 10000, Tax $>$ 0.04 |
| $Q_1^s$ | SELECT Nation FROM Supplier S |
| $Q_2^s$ | SELECT * FROM LineOrder LO, Customer C WHERE LO.CustKey = C.CustKey, 5000 $<$ CustKey $<$ 5100 |
| $Q_3^s$ | SELECT * FROM LineOrder LO, Supplier S WHERE LO.SuppKey = S.SuppKey, SuppKey $<$ 100, Tax $>$ 7 |
| $Q_4^s$ | SELECT * FROM Part P WHERE PartKey $<$ 10000, Size $>$ 10 |
| $Q_5^s$ | SELECT * FROM Date D WHERE Year = 1995 |
| $Q_6^s$ | SELECT * FROM LineOrder LO WHERE Revenue $>$ 7000000, SupplyCost $<$ 100000 |
| $Q_7^s$ | SELECT * FROM Customer C WHERE CustKey $>$ 24000, Region = 'Europe' |
| $Q_8^s$ | SELECT * FROM Supplier S WHERE SuppKey $<$ 30000, Nation = 'Brazil' |

### 6.1 Effect of Mechanism Parameter

The first set of experiments explores the effect of the parameter $a$ (in the tuple quality function) on the query quality $\kappa$ defined in Definition 4, on the real datasets *world* and *DBLP*.

Figure 2 depicts the change trend of the *average* query quality with the parameter of $a$ varying from 0.1 to 0.9. As expected, the higher the value of $a$, the higher the quality. The reason why the query quality on the *world* dataset is consistently better than that on the *DBLP* dataset is, the ratio of the number of tuples missing the *critical* attribute values (w.r.t. the uncertain lineage) to the total number of lineage tuples on *DBLP* dataset is relatively larger than that on *world* dataset. Without loss of generality, the value of $a$ is set as 0.5 in the rest of the experiments.

### 6.2 Active Strategy vs. Lazy Strategy

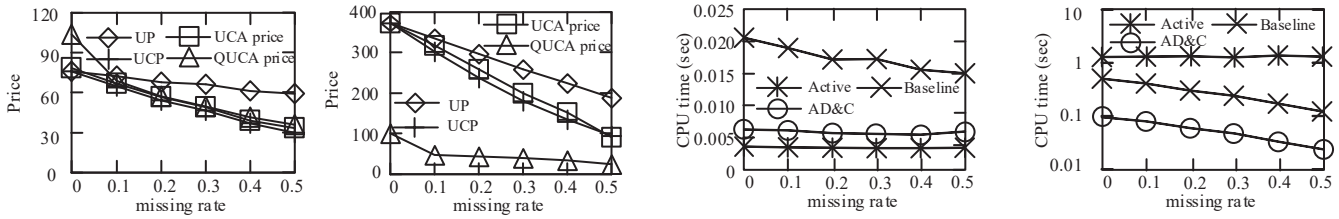
In this set of experiments, we study the performance of Active and Lazy (as described in Section 5.1). Figure 3 plots the average execution time of the two strategies for deriving lineage sets on the real datasets *world* and *DBLP*.

One can observe that, even in logarithmic scale, Active performs better than Lazy remarkably. The reason behind is that, for Active strategy, it suffices to derive lineage sets along with the query via a single dataset read. In contrast, Lazy strategy has to visit the whole dataset to find result tuples' lineage sets after getting the result tuple set. During the access to the dataset, for each tuple in the dataset, it needs to check whether it is a lineage tuple for each of the result tuples, and thus incurring more overhead. Moreover, it is sensitive to the size of result tuples. The larger the result tuple set, the larger the cost of Lazy, as analyzed in Section 5.1. We employ Active strategy to derive the lineage sets in the experiments.

### 6.3 Evaluation on Price Functions

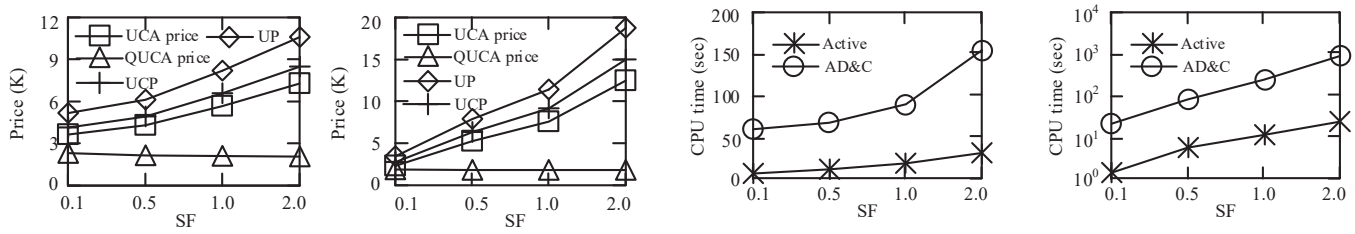
*Effect of missing rate*. First, we inspect the influence of the missing rate on the price (computation). Figure 4 depicts the corresponding experimental results over real datasets.

As plotted in Figure 4(a) and Figure 4(b), respectively, the reported price on every dataset is the average price of all the queries over the corresponding datasets (as listed in Table 5). One can observe that, with the increasing missing rate, the amount of each price on *world* and *DBLP* is dropping. It is due to the shrinking query result set. The amount of UP is particularly equal to the average number of query answers as shown in Table 6. It is important to point out that, the amount of QUCA price is incomparable to others, since the price coefficient $\Delta$ of the QUCA price function could be tuned by users (e.g., referring to the data market environment). While one can conclude that, the QUCA price is *more stable* than other prices with the growth of the missing rate. As the definition of QUCA price is insensitive to the query result size, QUCA price is able to avoid leaking information of result set (e.g., the scale of




(a) *world* (b) *DBLP* (c) *world* (d) *DBLP*

Fig. 4. Evaluation on *world* and *DBLP* datasets




(a) *TPC-H* (b) *SSB* (c) *TPC-H* (d) *SSB*

Fig. 5. Evaluation on *TPC-H* and *SSB* datasets

TABLE 6
The Average Number of Result Tuples Under Different Missing Rates

| Dataset | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 |
|---|---|---|---|---|---|---|
| *world* | 76 | 72 | 68 | 66 | 61 | 59 |
| *DBLP* | 372 | 335 | 297 | 259 | 225 | 188 |

TABLE 7
The Average Query Qualities Under Different Missing Rates

| Dataset | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 |
|---|---|---|---|---|---|---|
| *world* | 1.000 | 0.741 | 0.687 | 0.669 | 0.648 | 0.637 |
| *DBLP* | 1.000 | 0.502 | 0.501 | 0.500 | 0.500 | 0.500 |

result set), which makes it more practical. Compared to UCP and UCA prices, the uniform price, i.e., UP, is the highest one in each case, as it does not consider any factor of data contribution, data completeness, or query quality.

In addition, as stated in Definition 5, the QUCA price closely relies on the query quality, even though it considers the data contribution and data completeness. Therefore, it confirms the analysis in Section 4 that, the UCA price function (as well as UP and UCP) probably reveals the output information (e.g., the scale of result set). While the QUCA price is mostly likely to be unrelated to the result set scale. It is worthwhile to mention that, these price functions, including UP, UCP, UCA, and QUCA, are also applicable to complete datasets (w.r.t. that cases of the missing rate being zero in diagrams).

Table 7 shows the average query quality over two real datasets. One can observe that, when the missing rate is zero, i.e., the data is complete, the quality is equal to the value of one (i.e., the highest value). Then, when the missing rate grows, the quality is decreasing. This is because, there are more and more tuples with the missing critical attributes for the growth of the missing rate, which makes the query results struggle with more noise. It is worth noting that, the QUCA price function heavily depends on the query quality, as they have the same decreasing trend with the growth of missing rate. Moreover, the query with the lower quality is impossible to be assigned a relatively higher QUCA price by our pricing scheme. As a consequence, we can conclude that, in addition to the data contribution, both the degree of missing values in tuples (i.e., the missing rate) and the degree of missing result tuples (i.e., the query quality) are critical to distinguish the datasets for sale.

On the other hand, the average time cost of the algorithms is depicted in Figure 4(c) and Figure 4(d), respectively. Observed from the diagrams, Baseline algorithm needs the most time to derive the query UCA/QUCA price. In contrast, AD&C algorithm spends less time on getting the query UCA/QUCA price. In the figures, the CPU time of Active corresponds to the cost of deriving lineage sets (which is executed before the price computation step). In order to derive the query price, Baseline algorithm needs to evaluate all the possible query lineage sets via merging the lineage set of each result tuple. The superiority of AD&C lies that, it first chooses a lineage set (with the lowest price) for each result tuple, and then, it merges those lineage sets into a unique *query* lineage set, based on which, the query price is finally derived.

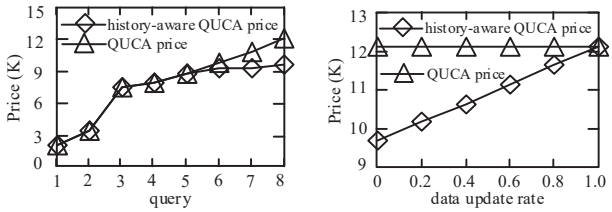
Moreover, for both algorithms, the time cost cuts down with the growth of the missing rate. This is because, the query result tuple set is becoming smaller, as true answers become less for the increasing missing rate, according to Definition 1. In particular, the average size of the result tuple set under each missing rate is reported in Table 6. We can also find that, the time cost of Active strategy closely depends on the dataset, since visiting the dataset is the dominant cost for Active strategy. In addition, the query price derived from AD&C and Baseline is identical for each dataset. This is partly because the query lineage set is unique in almost all cases for selection and join operators. For the projection operator, one tuple can only be projected to one result tuple, and thus, finding the lowest-price lineage set for each result tuple (by AD&C) is equivalent to finding the *query* lineage set with the lowest-price (i.e., the query price) (by Baseline). Hence, AD&C has a pretty high accuracy on price computation, even with the approximate ratio being one. It is much tighter than the theoretical approximate ratio analyzed in Section 5.2. We adopt AD&C algorithm to derive the query price in the rest of experiments.

TABLE 8
The Average Number of Result Tuples Under Different Scale Factors

| Dataset | 0.1 | 0.5 | 1 | 2 |
|---|---|---|---|---|
| *TPC-H* | 5,143 | 6,129 | 8,203 | 10,592 |
| *SSB* | 3,422 | 7,872 | 11,425 | 18,754 |

TABLE 9
The Average Query Qualities Under Different Scale Factors

| Dataset | 0.1 | 0.5 | 1 | 2 |
|---|---|---|---|---|
| *TPC-H* | 0.653 | 0.610 | 0.607 | 0.597 |
| *SSB* | 0.551 | 0.549 | 0.547 | 0.549 |




(a) Price comparison (b) Effect of data update

Fig. 6. Evaluation on the history-aware price

*Effect of data cardinality*. We evaluate the impact of data set size on the price (computation) via changing the scale factor (SF) from 0.1 to 2.0 over the two benchmarks *TPC-H* and *SSB*. Figure 5(a) and Figure 5(b) depict the average amount of each price function over queries on *TPC-H* and *SSB* benchmarks, respectively. It is obvious that, UP, UCP, and UCA price become increasingly higher with the growth of SF. This is because, the query result set as well as the lineage set turns bigger when the data scale changes larger, which could be conformed via Table 8. In particular, UP is still the highest price in each case.

Furthermore, one can observe that, the QUCA price is not very sensitive to the data scale, and thereby avoids leaking output information. The query quality is almost unrelated to the data scale, as reported in Table 9. The reason is that, the data scale has no direct relationship to the query quality, which makes sense in practice. We also report the average execution time of AD&C algorithm and Active strategy in Figure 5. Clearly, the execution time increases significantly with the growth of SF, since the query result tuple set (as reported in Table 8) and the corresponding lineage sets are becoming larger. In addition, Active consumes the relatively less time to derive lineage sets of each result tuple on benchmarks. Overall, the phenomenon that appeared in the benchmark datasets is similar to that over the real datasets. Our presented AD&C algorithm is capable of deriving the query price over large-scale datasets.

### 6.4 Evaluation on History-Aware Price Functions

First, we investigate the history-aware price on the bechmark *SSB*. In the experiments, we generate 8 query instances for the query $Q_6^s$ (SELECT * FROM LineOrder LO WHERE Revenue $>$ 7000000, SupplyCost $<$ 100000) on the *SSB* benchmark, with varying parameter values of Revenue and SupplyCost within their domains randomly, where we avoid the query instances with the empty result set.

Figure 6(a) shows the history-aware QUCA price with the consecutive process of the eight query instances as well as the (history-oblivious) QUCA price, where there is no tuple expired during the execution of these 8 query instances. Initially, at the initial phase, the history-aware QUCA price is equal to the (history-oblivious) QUCA price, since there is no overlap on the query lineage sets among the initial several query instances. Then, the gap between the history-aware QUCA price and the (history-oblivious) QUCA price appears at the sixth query instance and then becomes bigger and bigger. Even though only looking at the first eight instances, it suffices to understand that, the price gap will continue becoming increasingly bigger for more query instances. This is because, when processing more query instances, more tuples in the data set will be free. Thus, the price gap between QUCA and history-aware QUCA becomes larger.

Second, we study the effect of the data expired/update rate on the history-aware QUCA price. In the experiments, we generate a state label for each tuple in the datasets at a certain time stamp, and those state labels are updated at a certain update rate for each following time stamp. It means that, the corresponding tuples with updated lables are unavailable for the customers. That is to say, those tuples are not free for the customers that have ever paid for them. The update rate is equal to the ratio of the number of expired tuples to the total dataset size.

When the update rate changes from 0 (w.r.t. no update) to 1.0 (w.r.t. complete update), Figure 6(b) plots the history-aware QUCA price for those 8 instances of the query $Q_6^s$ on *SSB* benchmark. One can observe that, the history-aware QUCA price is consistently growing and approaching the (history-oblivious) QUCA price. Note that, when the update rate is zero, it indicates that there is no updated on the collected lineage tuples, and thereby all these lineage tuples are *active*, and can be re-used (in free of charge). Hence, at this time, the history-aware QUCA price is the lowest. With the ascending update rate, more and more tuples in the collected lineage set are expired, and thus cannot be re-used. If the update rate is equal to one, all tuples in the collected lineage set are expired. Thus, when new query instances are arriving, the purchased information of historical queries cannot be re-used. At this extreme case, the history-aware pricing is meaningless, and the history-aware QUCA price is identical to the (history-oblivious) QUCA price.

In summary, compared with the uniform (completeness-based) prices UP and UCP, the UCA price is well refined and more reasonable. The data contribution has a significant effect on the UCA price, so that this price is a bit sensitive to the query output size. Hence, although it is arbitrage-free, the UCA price might leak some output information, which is a defect in many real-life scenarios. In contrast, the QUCA price is closely related to the query result uncertainty, i.e., the query quality. Moreover, it is basically insensitive to the output scale. Meanwhile, a good query quality does deserve a higher QUCA price. One can conclude that, in addition to the data contribution, both the data completeness and the query quality can further value the data sets for sale. On the other hand, the history-aware price is cost-effective, in contrast to the history-oblivious price. In addition, Active strategy is more effective than Lazy strategy to derive the lineage sets. AD&C algorithm has good scalability for price computation on large datasets.

## 7 Related Work

In this section, we overview query processing over incomplete data and the data pricing mechanisms, respectively.

**Querying incomplete data**. The foundational research of incomplete data from the 1980s, first by Imielinski and Lipski [33] and then by Abiteboul, Kanellakis, and Grahne [34] provide models of incompleteness appropriate for handling queries in different relational languages, and establish the computational costs of the key tasks associated with those models. A series of index structures has been developed to organize incomplete data, such as the bit string-augmented R-tree (BR-tree) and the MOSAIC structure [35], bitmaps and quantization [36], for improving the query efficiency in high-dimensional incomplete databases. Also, several spatial queries over incomplete data have been investigated, such as skyline queries [7], [10], [11], [12], [37], ranking or top-$k$ queries [8], [38], [39], similarity queries [9], [31], and comprehensive incomplete data studies [40], [41].

**Data pricing schemes**. First, the query-based pricing model is a typical pricing solution in database community, in which the structural granularity of pricing is a query. The seller sets the price of a specific set of base queries (also called view), and then, the algorithm has to derive the price of any other query that could be made to the database. This group consists of pricing generalized chain queries (i.e., a special type of join conjunctive query when the base queries are only selection queries) [14], [42] and pricing SQL queries [13], [16]. There are some efforts on revenue maximizing arbitrage-free pricing [19], [43], [44], the pricing problem for trading time series data and personal data [45], [46], [47], [48], pricing queries while protecting the seller's privacy [49], [50] or in cloud environments [51]. In contrast, another kind of pricing scheme considers the tuples of relations as the structural granularity of the pricing function [15], [26], [52], [53]. It is usually based on the data lineage, i.e., the set of the tuples contributing to the result tuples of a query. Our proposed pricing scheme iDBPricer belongs to the category. QIRANA [16], [54], standing the viewpoint of the data buyer, employs the *possible world* semantic to price relational queries.

In addition, there is a series of studies on pricing *machine learning* tasks [17], [18], [55], pricing-related problems in both advertising markets and labor markets [56], etc. In the data acquisition problem [57] for correlation analysis, the quality, join informativeness, and price issues of data acquisition are considered. Overall, the study of pricing problem in data markets is an active research line. Note that, since the aforementioned existing studies do not consider data completeness, the prices derived by them do not change with varying missing rates. Thus, they are *incomparable* to our pricing methods for incomplete data. We believe that, more mature and robust pricing models for incomplete data will appear soon inspired by our pricing mechanism.

## 8 Conclusions

In this paper, for the first time, we propose a *practical pricing mechanism* iDBPricer for querying incomplete data. We present two novel query price functions, i.e., the UCA price and QUCA price, via considering the data contribution, data completeness, and query quality. We extend the price functions to tackle the history-aware pricing problem. Efficient algorithms are developed to calculate the query prices. Compared with the state-of-the-art price methods, extensive experiments with both real and benchmark data sets demonstrate the superioty of iDBPricer. The UCA price and QUCA price are more reasonable and practical. In the future, we intend to explore the pricing problems over probabilistic data and unstructured data.

**Acknowledgments.** This work was supported in part by the NSFC under Grants No. 61902343, 61972338, 61825205, and 61772459, the National Key Research and Development Program of China under Grants No. 2018YFB1004003 and 2017YFB1400603. Yunjun Gao is the corresponding author of the work.

## References

[1] "YoueData," [Online]. Available: http://www.youedata.com.
[2] "GBDEX," [Online]. Available: http://www.gbdex.com/.
[3] "GNIP APIs," [Online]. Available: http://support.gnip.com/apis/.
[4] "Xignite," [Online]. Available: https://www.xignite.com/.
[5] "Here," [Online]. Available: https://www.here.com/en.
[6] "Factual," [Online]. Available: https://www.factual.com/.
[7] M. E. Khalefa, M. F. Mokbel, and J. J. Levandoski, "Skyline query processing for incomplete data," in *ICDE*, pp. 556–565, 2008.
[8] P. Haghani, S. Michel, and K. Aberer, "Evaluating top-$k$ queries over incomplete data streams," in *CIKM*, pp. 877–886, 2009.
[9] W. Cheng, X. Jin, J.-T. Sun, X. Lin, X. Zhang, and W. Wang, "Searching dimension incomplete databases.," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 3, pp. 725–738, 2014.
[10] 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, 2016.
[11] K. Zhang, H. Gao, X. Han, Z. Cai, and J. Li, "Probabilistic skyline on incomplete data," in *CIKM*, pp. 427–436, 2017.
[12] K. Zhang, H. Gao, X. Han, Z. Cai, and J. Li, "Modeling and computing probabilistic skyline on incomplete data," *IEEE Trans. on Knowl. Data Eng.*, 2019.
[13] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu, "Toward practical query pricing with query market," in *SIGMOD*, pp. 613–624, 2013.
[14] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu, "Query-based data pricing," *Journal of the ACM*, vol. 62, no. 5, p. 43, 2015.
[15] P. Upadhyaya, M. Balazinska, and D. Suciu, "Price-optimal querying with data APIs," *Proceedings of the VLDB Endowment*, vol. 9, no. 14, pp. 1695–1706, 2016.
[16] S. Deep and P. Koutris, "QIRANA: A framework for scalable query pricing," in *SIGMOD*, pp. 699–713, 2017.
[17] R. Jia, D. Dao, B. Wang, F. A. Hubis, N. M. Gurel, B. Li, C. Zhang, C. Spanos, and D. Song, "Efficient task-specific data valuation for nearest neighbor algorithms," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, pp. 1610–1623, 2019.
[18] L. Chen, P. Koutris, and A. Kumar, "Towards model-based pricing for machine learning in a data marketplace," in *SIGMOD*, pp. 1535–1552, 2019.
[19] S. Chawla, S. Deep, P. Koutrisw, and Y. Teng, "Revenue maximization for query pricing," *Proceedings of the VLDB Endowment*, vol. 13, no. 1, pp. 1–14, 2019.
[20] E. F. Codd, "Extending the database relational model to capture more meaning," *ACM Trans. Database Syst.*, vol. 4, no. 4, pp. 397–434, 1979.
[21] S. Greco, C. Molinaro, and F. Spezzano, *Incomplete data and data dependencies in relational databases*, vol. 4 of *Synthesis Lectures on Data Management*. Morgan & Claypool Publishers, 2012.
[22] "Windows azure marketplace," [Online]. Available: http:// datamarket.azure.com/.
[23] Y. Cui and J. Widom, "Practical lineage tracing in data warehouses," in *ICDE*, pp. 367–378, 2000.
[24] F. Stahl and G. Vossen, "Data quality scores for pricing on data marketplaces," in *ACIIDS*, pp. 215–224, Springer, 2016.

[25] H. Yu and M. Zhang, "Data pricing strategy based on data quality," *Computers & Industrial Engineering*, vol. 112, pp. 1–10, 2017.
[26] R. Tang, H. Wu, B. Zhifeng, K. Thomas, and B. Stephane, "The price is right, The price of relational and probabilistic relational data," tech. rep., TRB5/12, 2012.
[27] "World," [Online]. Available: https://dev.mysql.com/doc/world-setup/en/.
[28] "DBLP," [Online]. Available: https://snap.stanford.edu/data/com-DBLP.html.
[29] "TPC-H," [Online]. Available: http://www.tpc.org/tpch.
[30] "SSB," [Online]. Available: http://www.cs.umb.edu/ poneil/SatrSchemaB.PDF.
[31] X. Miao, Y. Gao, G. Chen, B. Zheng, and H. Cui, "Processing incomplete $k$ nearest neighbor search," *IEEE Transactions on Fuzzy Systems*, vol. 24, no. 6, pp. 1349–1363, 2016.
[32] "AthenaPricing," [Online]. Available: https://aws.amazon.com/cn/athena/pricing/.
[33] T. Imieliński and W. Lipski, "Incomplete information in relational databases," *Journal of the ACM*, vol. 31, no. 4, pp. 761–791, 1984.
[34] S. Abiteboul, P. Kanellakis, and G. Grahne, "On the representation and querying of sets of possible worlds," *Theoretical Computer Science*, vol. 78, no. 1, pp. 159–187, 1991.
[35] B. C. Ooi, C. H. Goh, and K.-L. Tan, "Fast high-dimensional data search in incomplete databases," in *VLDB*, pp. 357–367, 1998.
[36] G. Canahuate, M. Gibas, and H. Ferhatosmanoglu, "Indexing incomplete databases," in *EDBT*, pp. 884–901, 2006.
[37] C. Lofi, K. El Maarry, and W.-T. Balke, "Skyline queries in crowd-enabled databases," in *EDBT*, pp. 465–476, 2013.
[38] K. Kolomvatsos, C. Anagnostopoulos, and S. Hadjiefthymiades, "A time optimized scheme for top-$k$ list maintenance over incomplete data streams," *Information Sciences*, vol. 311, pp. 59–73, 2015.
[39] M. A. Soliman, I. F. Ilyas, and S. Ben-David, "Supporting ranking queries on uncertain and incomplete data," *The VLDB Journal*, vol. 19, no. 4, pp. 477–501, 2010.
[40] Y. Gao and X. Miao, *Query processing over incomplete databases*, vol. 10 of *Synthesis Lectures on Data Management*. Morgan & Claypool Publishers, 2018.
[41] X. Miao, Y. Gao, S. Guo, and W. Liu, "Incomplete data management: A survey," *Frontiers of Computer Science*, pp. 1–22, 2017.
[42] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu, "Querymarket demonstration: Pricing for online data markets," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1962–1965, 2012.
[43] C. Xia and S. Muthukrishnan, "Arbitrage-free pricing in user-based markets," in *AAMAS*, pp. 327–335, 2018.
[44] W. Mao, Z. Zheng, and F. Wu, "Pricing for revenue maximization in iot data markets: An information design perspective," in *INFOCOM*, pp. 1837–1845, 2019.
[45] S. Sen, C. Joe-Wong, S. Ha, and M. Chiang, "Time-dependent pricing for multimedia data traffic: Analysis, systems, & trials," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 7, pp. 1504–1517, 2019.
[46] C. Niu, Z. Zheng, F. Wu, S. Tang, X. Gao, and G. Chen, "ERATO: Trading noisy aggregate statistics over private correlated data," *IEEE Trans. Knowl. Data Eng.*, 2019, online.
[47] C. Niu, Z. Zheng, S. Tang, X. Gao, and F. Wu, "Making big money from small sensors: Trading time-series data under pufferfish privacy," in *INFOCOM*, pp. 568–576, 2019.
[48] C. Niu, Z. Zheng, F. Wu, S. Tang, and G. Chen, "Online pricing with reserve price constraint for personal data markets," in *ICDE*, pp. 1978–1981, 2020.
[49] C. Li, D. Y. Li, G. Miklau, and D. Suciu, "A theory of pricing private data," *ACM Trans. Database Syst.*, vol. 39, no. 4, p. 34, 2014.
[50] C. Li and G. Miklau, "An adaptive mechanism for accurate query answering under differential privacy," *Proceedings of the VLDB Endowment*, vol. 5, no. 6, pp. 514–525, 2012.
[51] P. Upadhyaya, M. Balazinska, and D. Suciu, "How to price shared optimizations in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 6, pp. 562–573, 2012.
[52] R. Tang, D. Shao, S. Bressan, and P. Valduriez, "What you pay for is what you get," in *DEXA*, pp. 395–409, 2013.
[53] R. Tang, H. Wu, Z. Bao, B. Stephane, and P. Valduriez, "The price is right: Models and algorithms for pricing data," in *DEXA*, pp. 380–394, 2013.
[54] S. Deep, P. Koutris, and Y. Bidasaria, "QIRANA demonstration: Real time scalable query pricing," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1949–1952, 2017.
[55] L. Chen, P. Koutris, and A. Kumar, "Model-based pricing: Do not pay for more than what you learn!," in *DEEM*, pp. 1:1–1:4, 2017.
[56] C. Xia, *Pricing problems in online markets*. PhD thesis, Rutgers University-School of Graduate Studies, 2018.
[57] Y. Li, H. Sun, B. Dong, and H. W. Wang, "Cost-efficient data acquisition on online data marketplaces for correlation analysis," *Proceedings of the VLDB Endowment*, vol. 12, no. 4, pp. 362–375, 2018.

**Xiaoye Miao** received the PhD degree in computer science from Zhejiang University, China, in 2017. She is currently an assistant professor in the Center for Data Science, Zhejiang University, China. She was ever a research/postdoctoral fellow at University of New South Wales and City University of Hong Kong, respectively. Her research interests include uncertain/incomplete databases, data pricing, data cleaning, and graph management. She is a member of the IEEE and the CCF.

**Yunjun Gao** received the PhD degree in computer science from Zhejiang University, China, in 2008. He is currently a professor in the College of Computer Science, Zhejiang University, China. His research interests include database, big data management and analytics, and AI interaction with DB technology. He is a member of the ACM and the IEEE, and a senior member of the CCF.

**Lu Chen** received the PhD degree in computer science from Zhejiang University, China, in 2016. Prior to joining Aalborg University in 2017, she was a postdoctoral fellow in the School of Computer Science and Engineering, Nanyang Technological University, Singapore. She is currently an associate professor in Aalborg University, Denmark. Her research interests include metric data management and big data analytics.

**Huanhuan Peng** received the BS degree in software engineering science from Jilin University, China, in 2020. She is currently a PhD candidate in the Center for Data Science, Zhejiang University, China. Her research interests include crowdsoured data management and data pricing.

**Jianwei Yin** received the PhD degree in computer science from Zhejiang University, in 2001. He is currently a professor in the College of Computer Science, Zhejiang University. He is a visiting scholar at the Georgia Institute of Technology, in 2008. His research interests include service computing and data management.

**Qing Li** is a chair professor of Data Science, the Hong Kong Polytechnic University since Dec 2018. Prior to that, he has taught at City University of Hong Kong, the Hong Kong University of Science and Technology, and the Australian National University (Canberra, Australia). He is currently a Fellow of IET (formerly IEE), a Senior Member of IEEE, a member of ACM-SIGMOD and IEEE Technical Committee on Data Engineering. He is the Chairperson of the Hong Kong Web Society, and also served/is serving as an executive committee (EXCO) member of IEEE-Hong Kong Computer Chapter and ACM Hong Kong Chapter.