Ranking Desired Tuples by Database Exploration

Xuedi Qin†, Chengliang Chai†, Yuyu Luo†, Tianyu Zhao†, Nan Tang†, Guoliang Li†, Jianhua Feng‡, Xiang Yu†, Mourad Ouzzani‡
†Department of Computer Science, Tsinghua University ‡Qatar Computing Research Institute, HBKU
{qxd17@mails., chaicl15@mails., luoyy18@mails., zhaoty17@mails., liuguoliang@, fengjh@, x-yu17@mails.}@tsinghua.edu.cn,
{ntang, mouzzani}@hbku.edu.qa

Abstract—Database exploration – the problem of finding and ranking desired tuples – is important for data discovery and analysis. Precisely specifying SQL queries is not always feasible in practice, such as “finding and ranking off-road cars based on a combination of Price, Make, Model, Age, and Mileage.” – not only due to the query complexity (e.g., which may have many if-then-else, and, or and not logic), but also because the user typically does not have the knowledge of all data instances.

We propose DEXPLORER, a system for interactive database exploration. DEXPLORER offers a simple and user-friendly interface which allows to: (1) confirm whether a tuple is desired or not, and (2) decide whether a tuple is more preferred than another. Behind the scenes, we jointly use multiple ML models to learn from the above two types of user feedback. Moreover, in order to effectively involve users, we carefully select the set of tuples for which we need to solicit feedback. Therefore, we devise question selection algorithms that consider not only the estimated benefit of each tuple, but also the possible partial orders between any two suggested tuples. Experiments on real-world datasets show that DEXPLORER is more effective than existing approaches.

I. INTRODUCTION

We study the problem of interactive database exploration in scenarios where a user is interested in finding and ranking some tuples from a given dataset but where it is usually hard to precisely specify the intent. This is common in practice, for both non-experts who cannot write SQL queries and scripts, and experts who are not familiar with the data. Further complicating the problem, the query intent may be complex, e.g., finding tuples that satisfy a combination of many if-then-else, and, or and not conditions, and are ranked by a weighted function over multiple attributes.

Existing Works. We categorize existing works as follows.

Keyword search [15], [20], [6], [25], [23] retrieves desired tuples based on keywords. However, such queries are typically under-specified and ambiguous. Even with iterative refinement, it is impossible to formulate a keyword query that can be unambiguously translated to a (complex) query.

Query-by-example either infers an SQL query \( Q \) [22], [8] or learns a ML model \( M \) [17] over a database \( D \) using user provided examples \( E \). Based on \( E \), they try to infer \( Q \) or \( M \) such that \( Q(D) \) or \( M(D) \) is good w.r.t. \( E \), where the quantification of “goodness” varies in different applications. Preference between tuples approaches [21], [27] infer the ranking of tuples based on user provided partial orders of tuple pairs; [27] proposes an approach to find the top-1 result while [21] ranks all tuples.

Limitations of Existing Works. Let \( DE \) denote the general problem to discover and rank tuples, \( DE\)-Decision the special case of \( DE \) where users only want specific tuples (without ranking), and \( DE\)-Ranking another special case where users want the ranking over all tuples. Table I summarizes our comparison of different methods for data exploration.

(I) [Supported Problems.] Most approaches focus on either \( DE\)-Decision or \( DE\)-Ranking. Besides DEXPLORER, only SQLSynthesizer [28] can discover an SQL query with ranking, but it can only support simple (hierarchical) ranking functions such as \( \text{ORDER BY} \) attribute year first and then \( \text{ORDER by} \) attribute kilometer. No existing approach can support a combination of selection conditions and a more natural (but maybe more complicated) ranking function such as \((-0.018 \times \text{price} + 0.982 \times \text{powerPS})\).

(II) [Supported Special Cases.] Even for approaches that support either \( DE\)-Decision or \( DE\)-Ranking, some of them can only handle special cases, as described in the column “Supported Special Cases” in Table I.

(III) [Interactive vs. Static.] Some approaches are interactive while others support static input. Note that some back-end inference algorithms can easily support user interactions, such as ML models, while it might be hard with others, such as keyword-based or SQL query inference.

The main difference between this work and those existing works is how to holistically solve the general DE problem, instead of treating \( DE\)-Decision and \( DE\)-Ranking separately, by minimizing the interactions with the user.

Our Proposal. We propose DEXPLORER for with the following main features. [Interactivity.] DEXPLORER is designed as an interactive system, because providing a set of representative, unbiased, and sufficient samples in one shot is hard. [Usability.] It offers an easy-to-use interface for any user by mean of a list of tuples with two simple operations “click” and “drag”, for the user to provide true/false tuple labels and partial orders between tuples, respectively. [Capability.] In order to infer (possibly) complicated query intent, we propose to jointly train several ML models instead of guessing SQL queries, along the same line of [17], because tightly specifying conditions in SQL queries in database exploration might be
** Contributions.** Our contributions are summarized below. 
(1) We formally define the problem of interactive database exploration. (Section II)  
(2) We present DEXPlorer that iteratively trains ML models and predicts on-the-fly results for interactive data exploration with the users. (Section III)  
(3) We present a hybrid decision and ranking model for question selection to minimize human cost. (Section IV)  
(4) We conduct extensive experiments to show the effectiveness of DEXPlorer. (Section V)  

**II. PROBLEM STATEMENT**

Given a relational table $T$, the user wants to find a subset $R \subseteq T$, and tuples in $R$ are ranked. In practice, a user’s query intent might be formulated as either an SQL query, some machine learning (ML) models, or some specific logic. Generally speaking, we want to infer/learn a method/model $f()$ as the proxy of a user’s query intent, where $f(T)$ returns the ranked list $R$ that is close to $R$.

**Example 1:** Suppose a user wants a new manual petrol car that is produced after year 2010, not provided by commercial sellers, and its brand can be either BMW with price $\leq 10000$, or Volkswagen with price $\leq 8000$. Moreover, assume that she wants all cars to be ranked, e.g., using a weighted sum function: $-0.018 \times price + 0.982 \times powerPS$. The query intent could be expressed as Q1 below:

```
SELECT * FROM Car
WHERE: seller != "commercial" AND year >= 2010
AND gearbox="manually" AND fuelType="petrol"
AND ([brand = "bmw" AND price <= 10000] OR
([brand = "volkswagen" AND price <= 8000])
ORDER BY -0.018 * price + 0.982 * powerPS DESC;
```

Example 1 shows that a user’s query intent may be: (1) **hard to specify:** there are complicated predicates in the WHERE clause and it is almost impossible to manually specify the weighted sum function with correct parameters in the ORDER BY clause, and (2) **hard to infer:** no existing work (see Table I) can effectively infer such a query, not to say more complicated cases.

**User Operations.** We allow two user operations:
(1) true/false labeling: the user may label a given tuple as true(desired) or false(not desired); and  
(2) a partial order: given two tuples $t_i$ and $t_j$, the user might tell which tuple is more preferable.

**User Questions.** A question $Q$ is a list of $k$ tuples, which solicits two types of labels from a user:
(1) decision: the user will split $Q$ into three disjoint sets of tuples: $\mathbb{D}^+$ with true labels, $\mathbb{D}^-$ with false labels, and $\mathbb{D}^0$ with unknown labels. Let $\mathbb{D}$ be the set of all tuple labels, i.e., $\mathbb{D} = \mathbb{D}^+ \cup \mathbb{D}^- \cup \mathbb{D}^0$.  
(2) ranking: rank (partial) pairs of tuples in $\mathbb{D}^+$, which gets a set $\mathbb{R}$ of partial orders. Implicitly, any tuple $t_j \in \mathbb{D}^+$ is more preferred than any tuple $t_i \in \mathbb{D}^-$, and there is no need to rank two tuples in $\mathbb{D}^0$, denoted by $t_x \equiv t_y$ for any $t_x, t_y \in \mathbb{D}^0$.

**Example 2:** The table in Figure 1 shows the labeling examples based on $Q_1$ in Example 1. (1) The user can annotate the tuples she desires or not, e.g., $\mathbb{D}^+ = \{t_1, t_2, t_3, t_4\}$ and $\mathbb{D}^- = \{t_5, t_6\}$. (2) The user can specify which cars are more preferred for tuples in $\mathbb{D}^+$, either through a total order such as $t_1 \succ t_2 \succ t_3 \succ t_4$ or a set of partially ordered lists such as $t_1 \succ t_2 \succ t_3$ and $(t_1 \succ t_4)$.

**Problem Statement.** Given a table $T$, a parameter $k$ (i.e., the number of tuples to show in each question), and a budget $n$ for the number of questions, the problem of interactive database exploration (IDE) is to iteratively ask $n$ questions $\{I_1, \ldots, I_n\}$ ($|I_i| = k$ for $i \in [1, n]$) to the user, and infer a ranked set $R$ of tuples based on the user feedback.

**Two Research Challenges.** There are two main research challenges to solve IDE: (1) **Answer Inference:** how to infer $R$ based on the user feedback for $I_1, \ldots, I_n$? (Section III) (2) **Question Selection:** how to select a question $I_i$ in the $i$-th iteration? (Section IV)

**III. OVERVIEW OF DEXPlorer**

**A. System Overview (Figure 1)**

**Front-end.** The user interacts with DEXPlorer in multiple iterations until the user budget is used up or the answer cannot be improved. At each iteration, the user is provided with a question $I$ with $k$ tuples, on which two operations are permissible: (1) “click” to annotate a tuple as either true or false; and (2) “drag” to rank a tuple higher than

<table>
<thead>
<tr>
<th>Project</th>
<th>Input</th>
<th>DE-Decision</th>
<th>DE-Ranking</th>
<th>DE</th>
<th>Supported Special Cases</th>
<th>Interactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISCOVER [16]</td>
<td>(1)</td>
<td>✓</td>
<td></td>
<td></td>
<td>only find a small number of interesting tuples</td>
<td>✓</td>
</tr>
<tr>
<td>SQLSynthesizer [26]</td>
<td>(2)</td>
<td></td>
<td>✓</td>
<td></td>
<td>only simple ranking function</td>
<td>✓</td>
</tr>
<tr>
<td>SQUID [18]</td>
<td>(3)</td>
<td>✓</td>
<td></td>
<td></td>
<td>does not support the “or” predicate</td>
<td>✓</td>
</tr>
<tr>
<td>AIDE [5]</td>
<td>(4)</td>
<td>✓</td>
<td></td>
<td></td>
<td>does not support the “or” predicate</td>
<td>✓</td>
</tr>
<tr>
<td>Huang et al. [17]</td>
<td>(5)</td>
<td>✓</td>
<td></td>
<td></td>
<td>only rank categorical attributes</td>
<td>✓</td>
</tr>
<tr>
<td>Chaudhuri et al. [2]</td>
<td>(6)</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Qian et al. [21]</td>
<td>(7)</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Xie et al. [27]</td>
<td>(8)</td>
<td>✓</td>
<td></td>
<td></td>
<td>only find top-1 result</td>
<td>✓</td>
</tr>
<tr>
<td>DEXPlorer</td>
<td>(1,2,3)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**TABLE I**

**COMPARISION WITH STATE-OF-THE-ART (USER INPUT 1: KEYWORD, 2: EXAMPLES, 3: PARTIAL ORDERS)**

hard. More specifically, we use random forests [19] to infer the desired tuples, and a hybrid ranking model combining LambdaMART [26] and Ranking SVM to rank tuples. [Effectiveness.] We devise novel question selection algorithms that jointly estimate the benefit of soliciting feedback from both tuple labeling and partial order labeling.

Project | Input | DE-Decision | DE-Ranking | DE | Supported Special Cases | Interactive |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DISCOVER [16]</td>
<td>(1)</td>
<td>✓</td>
<td></td>
<td></td>
<td>only find a small number of interesting tuples</td>
<td>✓</td>
</tr>
<tr>
<td>SQLSynthesizer [26]</td>
<td>(2)</td>
<td></td>
<td>✓</td>
<td></td>
<td>only simple ranking function</td>
<td>✓</td>
</tr>
<tr>
<td>SQUID [18]</td>
<td>(3)</td>
<td>✓</td>
<td></td>
<td></td>
<td>does not support the “or” predicate</td>
<td>✓</td>
</tr>
<tr>
<td>AIDE [5]</td>
<td>(4)</td>
<td>✓</td>
<td></td>
<td></td>
<td>does not support the “or” predicate</td>
<td>✓</td>
</tr>
<tr>
<td>Huang et al. [17]</td>
<td>(5)</td>
<td>✓</td>
<td></td>
<td></td>
<td>only rank categorical attributes</td>
<td>✓</td>
</tr>
<tr>
<td>Chaudhuri et al. [2]</td>
<td>(6)</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Qian et al. [21]</td>
<td>(7)</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Xie et al. [27]</td>
<td>(8)</td>
<td>✓</td>
<td></td>
<td></td>
<td>only find top-1 result</td>
<td>✓</td>
</tr>
<tr>
<td>DEXPlorer</td>
<td>(1,2,3)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
three reasons: 

- RF, a collection of many DTs, are more robust at capturing complicated cases (e.g., with if-then-else, and, or and not) [24] and preventing overfitting, compared with DTs.
- In some cases, RF, with shallow decision trees and a few branches, are still interpretable.
- As will be seen shortly in question selection (Section IV), RF, which outputs a vector with 0’s and 1’s (each DT in an RF will output either 0 or 1 for one input), are a better choice than an SVM that gives a single value for computing diversity between tuples, an important feature for question selection.

**Ranking Inference.** We support linear weighted ranking functions because in many applications, especially in databases with numeric values, a weighted sum among multiple attributes is widely used to model user preferences, which is also observed by [18], [21].

The ranking problem in DEXPLORER is to learn a function \( g(t) \). We employ Ranking SVM [18], [21] as the basic model for two reasons: (1) The scoring function used to rank tuples can be roughly captured by a linear function, i.e., \( g(t) = \Vec{w} \cdot \Vec{t} \), where \( \Vec{w} \) is a weight vector quantifying the preference and importance of attributes, and \( \Vec{t} \) is the feature vector of tuple \( t \). Ranking SVM is a natural choice for this case. (2) Compared with complicated model like RankNet [1], Ranking SVM can avoid overfitting by using a small number of training data. However, one drawback of using a lightweight model, such as Ranking SVM, is that it needs sophisticated feature engineering. One way to combat this is to use another model to capture more distinguishing features, which is inspired by a classical solution for Click-Through Rate (CTR) Prediction.

**LambdaMART + Ranking SVM.** Inspired by the GBDT + LR model used in many commercial IR systems [13], [14], we propose a hybrid ranking model: LambdaMART + Ranking SVM, where LambdaMART [26] is a tree structure model based on MART (Multiple Additive Regression Tree) [9] that transforms the input features. The “Lambda” in LambdaMART denotes a special value which is the negative gradient in the MART algorithm. The output of LambdaMART is then fed to Ranking SVM to infer tuple ranking.

**C. Iteratively Training and Predicting.**

Given \( D_i \) for decision labels and \( R_i \) for partial order labels in the \( i \)-th iteration, we describe how to use them for decision and ranking training and prediction.

**Training.** (1) **Decision inference using RF:** we incrementally train RF by feeding new labels in \( D_i \), obtaining \( RF_i \). (2) **Ranking inference using LambdaMART + Ranking SVM:** Given a set of features of ordered tuple pairs \( R_i \) as training examples: (2.a) We first use \( R_i \) to incrementally train the LambdaMART model, obtaining \( LM_i \); (2.b) For each tuple \( t \) (whose one-hot encoding feature vector is denoted as \( \Vec{t} \) appearing in \( R_i \), \( LM_i(t) \) outputs a \( m \)-dimension transformed feature vector \( \{ x_j \} \), where \( m \) is the number of trees in LambdaMART (see Figure 1), and \( x_j \) \( (j \in [1,m]) \) is the leaf node index of \( \Vec{t} \) that ends up falling in the \( j \)-th
We concatenate output by the LambdaMART sub-model (LambdaMART to measure the similarity of two tuples. Let \( \bar{\mathbf{v}} \) be the concatenation (\( \circ \)) of \( \mathbf{v} \) and \( \mathbf{v'} \). For each ordered tuple pair \((t_i, t_j)\) in \( \mathcal{R}_i \), we use the enriched features \((\bar{\mathbf{v}}_i, \bar{\mathbf{v}}_j)\) to incrementally train Ranking SVM, obtaining \( \mathcal{R}_S \).

**Prediction.**
1. **Decision prediction:** we use \( RF \) to predict each unlabeled tuple, predicting \( \mathcal{R}_i \) as a set of desired tuples.
2. **Ranking prediction:** For each \( t \in \mathcal{R}_i \); (2.a) use the trained \( LM_i \) to get the enriched features \( \bar{\mathbf{v}} \); (2.b) compute \( t \) a score using the learned weight function \( \bar{\mathbf{w}} \) in \( \mathcal{R}_S \) as \( \bar{\mathbf{w}} \cdot \bar{\mathbf{v}} \); and (2.c) rank tuples in \( \mathcal{R}_i \) based on their scores.

### IV. QUESTION SELECTION

In each user iteration, we need to select a list of \( k \) tuples from the table \( T \) as one question \( I \).

#### A. Uncertainty and Diversity

Uncertainty is the most commonly used criteria for tuple selection; it measures the confidence of the current model on evaluating a question.

**Uncertainty for Decision Questions.** We define the uncertainty of a tuple \( t \) as the entropy of the predicted results of all decision trees in the random forest. Suppose we have \( n \) trees, and \( m \) is the number of trees which predicted the tuple \( t \) as positive, then the uncertainty of \( t \) is defined as:

\[
\text{u}(t) = - (e \log e + (1-e) \log (1-e)),
\]

where \( e = \frac{m}{n} \). The larger the \( u(t) \), the larger the uncertainty.

**Uncertainty for Ranking Questions.** Given a pair of tuples \( t_i \) and \( t_j \), the Ranking SVM model uses a hyperplane to make decisions about whether \( t_i > t_j \). It also learns a parameter vector \( \bar{\mathbf{w}} \), such that \( \bar{\mathbf{w}} \cdot \bar{\mathbf{v}}_i \) denotes the ranking score of tuple \( t_i \). The higher the score is, the higher the ranking \( t_i \) should be. If the pair is above the hyperplane, i.e., \( \bar{\mathbf{w}} \cdot \bar{\mathbf{v}}_i - \bar{\mathbf{w}} \cdot \bar{\mathbf{v}}_j > 0 \), then \( t_i > t_j \), and vice versa. Thus, those pairs close to the hyperplane are the uncertain ones, i.e., when \( |\bar{\mathbf{w}} \cdot \bar{\mathbf{v}}_i - \bar{\mathbf{w}} \cdot \bar{\mathbf{v}}_j| \) is close to 0. Besides selecting tuples that the decision and ranking models are uncertain about, we should also care about the diversity of the selected tuples. Consider two tuples with high uncertainty but with highly similar content, it is a waste of human efforts to label both of them because labeling only one of them can produce a model that is very likely to predict the other correctly.

**Diversity.** A simple way to measure diversity is to compute the string similarity of each tuple pair using some predefined function. However, such method does not consider the semantic information (dynamically) incorporated by the user feedback. Thus, we can use the feature vectors produced by RF and LambdaMART to measure the similarity of two tuples. Let \( \bar{\mathbf{v}}(t) \) be the prediction vector of all trees in RF for tuple \( t \), where the \( i \)-th element in \( \bar{\mathbf{v}}(t) \) denotes the prediction label (1 or 0) of the \( i \)-th tree in the random forest. Let \( \bar{\mathbf{v}}'(t) \) be the one-hot encoding transformed feature vector of tuple \( t \) by the LambdaMART sub-model (i.e., \( \bar{\mathbf{v}}'(t) = \mathbf{v}'(t) \)). We concatenate \( \bar{\mathbf{v}}(t) \) and \( \bar{\mathbf{v}}'(t) \) as \( \bar{\mathbf{v}}(t) \), which is used to compute the diversity. More specifically, we define the similarity \( s(t_i, t_j) \) of tuple \( t_i \) and \( t_j \) using the Cosine Similarity:

\[
s(t_i, t_j) = \cos(\bar{\mathbf{v}}(t_i), \bar{\mathbf{v}}(t_j)).
\]

The smaller the similarity, the larger the diversity between two tuples.

Considering both the uncertainty and diversity, we define the IDE question selection problem as follows.

**Definition 1 (IDE Question Selection):** Given a partially trained RF model and a hybrid ranking model, a table \( T \), and a number \( k \), select a set of \( k \) tuples \( S^* \) from \( T \) such that the following equation is minimized:

\[
s^* = \arg\min_{S \subseteq T, |S|=k} \sum_{t \in S} (1-u(t)) + \alpha \sum_{t_i, t_j \in S} |\bar{\mathbf{w}} \cdot \bar{\mathbf{v}}_i - \bar{\mathbf{w}} \cdot \bar{\mathbf{v}}_j| + \beta \sum_{s(t_i, t_j)} s(t_i, t_j)
\]

Theorem 1: The IDE question selection is NP-hard. We can prove that the IDE question selection problem is NP-hard by a reduction from the MAXSUMDISPERSION problem that is known to be NP-hard [11].

#### B. Question Selection Algorithms

1) **AQS: An Approximate Algorithm:** Inspired by [10], we present a 2-approximation algorithm, denoted by AQS, for IDE question selection problem. The algorithm first initializes an empty result set and calculates a new similarity \( s'(t_i, t_j) = \frac{1}{1+|\bar{\mathbf{w}} \cdot \bar{\mathbf{v}}_i - \bar{\mathbf{w}} \cdot \bar{\mathbf{v}}_j|} + \alpha |\bar{\mathbf{w}} \cdot \bar{\mathbf{v}}_i - \bar{\mathbf{w}} \cdot \bar{\mathbf{v}}_j| + \beta s(t_i, t_j) \) for each tuple pair \((t_i, t_j)\). It then selects the tuple pair that has not been added to the result set and with the least \( s' \) scores, and adds the pair to the result set. It iteratively runs \( \frac{k}{2} \) times. It adds another tuple to the result set if \( k \) is odd and the current result has \( k-1 \) tuples. The selected \( k \) tuples are then returned.

The time complexity of AQS is \( O(\max(|T|^2L, |T|^2k)) \), where \( L \) is the length of tuples. However, the time to compute a question can still be long when \( |T| \) is large, thus the above algorithm is not efficient enough to satisfy the online question selection requirement.

2) **IQS: An Efficient Algorithm:** We propose an efficient algorithm IQS (Algorithm 1) to solve the IDE question, which can return approximate results of \( S^* \) in interactive time. The main idea is that to first select the tuple with the highest decision uncertainty score, and then add it to the result set if this tuple has low similarity and high ranking uncertainty with the current selected tuples.

We first sort \( T \) by descending order of \( u(t) \) (line 1), and initialize the result set \( S \) as empty (line 2). Next, we iterate over each tuple \( t_i \) in \( T \), and check whether \( t_i \) has high similarity and low ranking uncertainty with respect to the current tuples in \( S \) (line 4 - 8). If yes, we just drop it; otherwise we add \( t_i \) into \( S \). Finally, \( S \) is returned as the selected question with \( k \) tuples (line 11).

**Complexity:** The time complexity for sorting \( T \) is \( O(|T| \log |T|) \). The time complexity to check whether \( t_i \) can be added to the result set is \( O(kL) \), where \( L \) is
The Ground Truth SQL Queries. We use SQL queries as the proxy for user’s query intent, because providing other types of ground truth would be subjective and thus not practical. We test 10 SQL queries $Q_1$-$Q_{10}$ on Car and $Q_{11}$-$Q_{20}$ on Pub. These queries are designed to be representative and cover many different cases. Due to the space limit, we omit more descriptions of these queries. For more details, please refer to our technical report (https://github.com/Qinxuedi/dexplorer).

V. Experiments

Datasets. We use two datasets: Car and Pub (for publications). The Car dataset is from Kaggle (https://www.kaggle.com/oregoleka/used-cars-database) with one relational table, and the Pub dataset is crawled from the ACM Digital Library (https://www.acm.org/publications/digital-library) and contains six relational tables. Table II gives more statistics about these two datasets.

The total number of tuple pairs whose predicted labels are different from ground truth labels. We use the normalized Kendall tau distance as the evaluation metric, denoted by $\text{accuracy} = \frac{2 \cdot K D}{N \cdot (N - 1)}$.

Comparisons. We test two flavors of DEXPLORER: (1) DEXPLORER-IQS uses the model in Section III-B for answer inference and IQS for question selection and (2) DEXPLORER-AQS uses the same answer inference model with DEXPLORER but different question selection algorithm AQS. We also compare DEXPLORER to (3) SQLSynthesizer [28] (denoted by SSY), which infers SQL by accepting ranked positive tuples.

Figure 2 shows the accuracy of $Q_1$-$Q_{10}$ for DEXPLORER-IQS, DEXPLORER-AQS. SSY. Figure 2 shows that: (1) With the increasing of #-Questions, the accuracy of DEXPLORER-IQS, DEXPLORER-AQS and SSY all increase. They finally achieve an accuracy of 0.916, 0.968, 0.499 respectively on average for $Q_1$-$Q_{10}$ after asking 20 questions. DEXPLORER-IQS behaves similarly to DEXPLORER-AQS, but DEXPLORER-AQS takes longer time (i.e., more than 1 hour for $Q_1$-$Q_{20}$), while DEXPLORER-IQS can return results in interactive time. (2) SSY supports decision using a decision tree and supports ranking by one or more attributes hierarchically. Thus, SSY achieves an accuracy of 1.0 on $Q_1$, $Q_{10}$, which contains simple decision problems and hierarchical ranking functions. But SSY behaves badly for complex decision and ranking problems, i.e., $Q_2$-$Q_5$, $Q_7$, $Q_8$, while DEXPLORER-IQS has a stable performance for all cases.

B. Efficiency

We also test the efficiency of DEXPLORER on IDE problems $Q_1$-$Q_{10}$. We repeat all experiments ten times to compute the average results. Table III shows the running time of answer inference and question selection by algorithms IQS and AQS.

We make the following observations: (1) IQS significantly outperforms AQS on all IDE problems (i.e., $Q_1$-$Q_{10}$). This observation also matches the result of the time complexity discussion in Section IV. (2) The results on answer inference and question selection of the tested algorithms do not vary a lot for different IDE problems on the same dataset. The reason is that the algorithms are independent of the complexity of the SQL queries. However, the complex SQL queries usually take more rounds to converge. (3) It takes less than 2 seconds on dataset Car to infer answers and select questions for the next round using IQS, which should be acceptable in practice. For the small dataset Pub, it only takes $\sim$0.1 second.
Acknowledgement. The work was supported by NSF of China (61925205, 61632016), Huawei, and TAL Education.

REFERENCES

VI. CONCLUSION

This paper presented DEXPLORE, an interactive database exploration system. DEXPLORE is based on a well-established answer inference model that selects a set of tuples to be annotated for relevance and partially ranked for preference by the user. It offers a user-friendly front-end that allows the user to select and rank tuples. We have proved that the optimal question selection problem is NP-hard and proposed an efficient and effective algorithm. Extensive experiments show the effectiveness of DEXPLORE.


