# Dynamic Query Forms for Database Queries

Liang Tang, Tao Li, Yexi Jiang, and Zhiyuan Chen

**Abstract**—Modern scientific databases and web databases maintain large and heterogeneous data. These real-world databases contain over hundreds or even thousands of relations and attributes. Traditional predefined query forms are not able to satisfy various ad-hoc queries from users on those databases. This paper proposes DQF, a novel database query form interface, which is able to dynamically generate query forms. The essence of DQF is to capture a user's preference and rank query form components, assisting him/her to make decisions. The generation of a query form is an iterative process and is guided by the user. At each iteration, the system automatically generates ranking lists of form components and the user then adds the desired form components into the query form. The ranking of form components is based on the captured user preference. A user can also fill the query form and submit queries to view the query result at each iteration. In this way, a query form could be dynamically refined till the user satisfies with the query results. We utilize the expected F-measure for measuring the goodness of a query form. A probabilistic model is developed for estimating the goodness of a query form in DQF. Our experimental evaluation and user study demonstrate the effectiveness and efficiency of the system.

**Index Terms**—Query Form, User Interaction, Query Form Generation,

## 1 Introduction

Query form is one of the most widely used user interfaces for querying databases. Traditional query forms are designed and predefined by developers or DBA in various information management systems. With the rapid development of web information and scientific databases, modern databases become very large and complex. In natural sciences, such as genomics and diseases, the databases have over hundreds of entities for chemical and biological data resources [22] [13] [25]. Many web databases, such as Freebase and DBPedia, typically have thousands of structured web entities [4] [2]. Therefore, it is difficult to design a set of static query forms to satisfy various ad-hoc database queries on those complex databases.

Many existing database management and development tools, such as EasyQuery [3], Cold Fusion [1], SAP and Microsoft Access, provide several mechanisms to let users create customized queries on databases. However, the creation of customized queries totally depends on users' manual editing [16]. If a user is not familiar with the database schema in advance, those hundreds or thousands of data attributes would confuse him/her.

### 1.1 Our Approach

In this paper, we propose a Dynamic Query Form system: DQF, a query interface which is capable of dynamically generating query forms for users. Different from traditional document retrieval, users in database retrieval are often willing to perform many rounds of actions (i.e., refining query conditions) before identifying the final candidates [7]. The essence of DQF is to capture user interests during user interactions and to adapt the query form iteratively. Each iteration consists of two types of user interactions: Query Form Enrichment and Query Execution (see Table 1). Figure 1 shows the work-flow of DQF. It starts with a basic query form which contains very few primary attributes of the database. The basic query form is then enriched iteratively via the interactions between the user and our system until the user is satisfied with the query results. In this paper, we mainly study the ranking of query form components and the dynamic generation of query forms.

TABLE 1
Interactions Between Users and DQF

| | |
|---|---|
| Query Form Enrichment | 1) DQF recommends a ranked list of query form components to the user.<br>2) The user selects the desired form components into the current query form. |
| Query Execution | 1) The user fills out the current query form and submit a query.<br>2) DQF executes the query and shows the results.<br>3) The user provides the feedback about the query results. |

### 1.2 Contributions

Our contributions can be summarized as follows:

- We propose a dynamic query form system which generates the query forms according to the user's desire at run time. The system provides a solution


- *Liang Tang, Tao Li, and Yexi Jiang are with School of Computer Science, Florida International University, Miami, Florida, 33199, U.S.A. E-mail: {ltang002,taoli, yjian004}@cs.fiu.edu*
- *Zhiyuan Chen is with Information Systems Department, University of Maryland Baltimore County, Baltimore, MD, 21250, U.S.A*

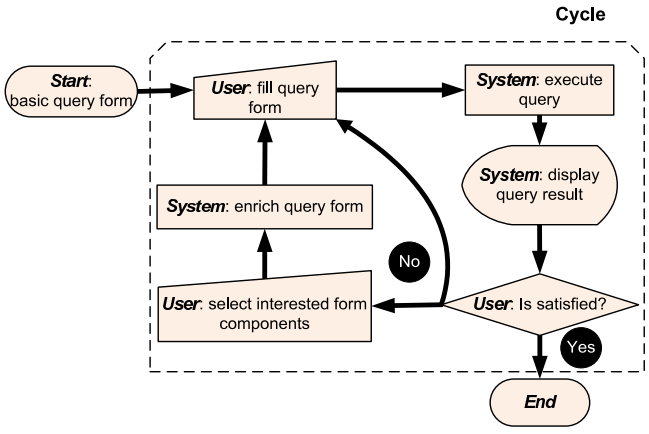



Fig. 1. Flowchart of Dynamic Query Form

for the query interface in large and complex databases.

- We apply F-measure to estimate the goodness of a query form [30]. F-measure is a typical metric to evaluate query results [33]. This metric is also appropriate for query forms because query forms are designed to help users query the database. The goodness of a query form is determined by the query results generated from the query form. Based on this, we rank and recommend the potential query form components so that users can refine the query form easily.
- Based on the proposed metric, we develop efficient algorithms to estimate the goodness of the projection and selection form components. Here efficiency is important because DQF is an online system where users often expect quick response.

The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 defines the query form and introduces how users interact with our dynamic query form. Section 4 defines a probabilistic model to rank query form components. Section 5 describes how to estimate the ranking score. Section 6 reports experimental results, and finally Section 7 concludes the paper.

## 2 Related Work

How to let non-expert users make use of the relational database is a challenging topic. A lot of research works focus on database interfaces which assist users to query the relational database without SQL. QBE (Query-By-Example) [36] and Query Form are two most widely used database querying interfaces. At present, query forms have been utilized in most real-world business or scientific information systems. Current studies and works mainly focus on how to generate the query forms.

**Customized Query Form:** Existing database clients and tools make great efforts to help developers design and generate the query forms, such as EasyQuery [3], Cold Fusion [1], SAP, Microsoft Access and so on. They provide visual interfaces for developers to create or customize query forms. The problem of those tools is that, they are provided for the professional developers who are familiar with their databases, not for end-users [16]. [17] proposed a system which allows end-users to customize the existing query form at run time. However, an end-user may not be familiar with the database. If the database schema is very large, it is difficult for them to find appropriate database entities and attributes and to create desired query forms.

**Automatic Static Query Form:** Recently, [16] [18] proposed automatic approaches to generate the database query forms without user participation. [16] presented a data-driven method. It first finds a set of data attributes, which are most likely queried based on the database schema and data instances. Then, the query forms are generated based on the selected attributes. [18] is a workload-driven method. It applies clustering algorithm on historical queries to find the representative queries. The query forms are then generated based on those representative queries. One problem of the aforementioned approaches is that, if the database schema is large and complex, user queries could be quite diverse. In that case, even if we generate lots of query forms in advance, there are still user queries that cannot be satisfied by any one of query forms. Another problem is that, when we generate a large number of query forms, how to let users find an appropriate and desired query form would be challenging. A solution that combines keyword search with query form generation is proposed in [12]. It automatically generates a lot of query forms in advance. The user inputs several keywords to find relevant query forms from a large number of pre-generated query forms. It works well in the databases which have rich textual information in data tuples and schemas. However, it is not appropriate when the user does not have concrete keywords to describe the queries at the beginning, especially for the numeric attributes.

**Autocompletion for Database Queries:** In [26], [21], novel user interfaces have been developed to assist the user to type the database queries based on the query workload, the data distribution and the database schema. Different from our work which focuses on query forms, the queries in their work are in the forms of SQL and keywords.

**Query Refinement:** Query refinement is a common practical technique used by most information retrieval systems [15]. It recommends new terms related to the query or modifies the terms according to the navigation path of the user in the search engine. But for the database query form, a database query is a structured relational query, not just a set of terms.

**Dynamic Faceted Search:** Dynamic faceted search is a type of search engines where relevant facets are presented for the users according to their navigation paths [29] [23]. Dynamic faceted search engines

are similar to our dynamic query forms if we only consider *Selection* components in a query. However, besides *Selections*, a database query form has other important components, such as *Projection* components. *Projection* components control the output of the query form and cannot be ignored. Moreover, designs of *Selection* and *Projection* have inherent influences to each other.

**Database Query Recommendation:** Recent studies introduce collaborative approaches to recommend database query components for database exploration [20] [9]. They treat SQL queries as items in the collaborative filtering approach, and recommend similar queries to related users. However, they do not consider the goodness of the query results. [32] proposes a method to recommend an alternative database query based on results of a query. The difference from our work is that, their recommendation is a complete query and our recommendation is a query component for each iteration.

**Dynamic Data Entry Form:** [11] develops an adaptive forms system for data entry, which can be dynamically changed according to the previous data input by the user. Our work is different as we are dealing with database query forms instead of data-entry forms.

**Active Feature Probing:** Zhu et al. [35] develop the active featuring probing technique for automatically generating clarification questions to provide appropriate recommendations to users in database search. Different from their work which focuses on finding the appropriate questions to ask the user, DQF aims to select appropriate query components.

## 3 Query Form Interface

### 3.1 Query Form

In this section we formally define the query form. Each query form corresponds to an SQL query template.

*Definition 1:* A query form $F$ is defined as a tuple ($\mathcal{A}_F$, $\mathcal{R}_F$, $\sigma_F$, $\bowtie(\mathcal{R}_F)$), which represents a database query template as follows:

$$F = \begin{array}{l}(\text{SELECT } A_1, A_2, ..., A_k \\ \text{FROM } \bowtie(\mathcal{R}_F) \text{ WHERE } \sigma_F),\end{array}$$

where $\mathcal{A}_F = \{A_1, A_2, ..., A_k\}$ are $k$ attributes for projection, $k > 0$. $\mathcal{R}_F = \{R_1, R_2, ..., R_n\}$ is the set of $n$ relations (or entities) involved in this query, $n > 0$. Each attribute in $\mathcal{A}_F$ belongs to one relation in $\mathcal{R}_F$. $\sigma_F$ is a conjunction of expressions for selections (or conditions) on relations in $\mathcal{R}_F$. $\bowtie(\mathcal{R}_F)$ is a join function to generate a conjunction of expressions for joining relations of $\mathcal{R}_F$.

In the user interface of a query form $F$, $\mathcal{A}_F$ is the set of columns of the result table. $\sigma_F$ is the set of input components for users to fill. Query forms allow users to fill parameters to generate different queries. $R_F$ and $\bowtie(\mathcal{R}_F)$ are not visible in the user interface, which are usually generated by the system according to the database schema. For a query form $F$, $\bowtie(\mathcal{R}_F)$ is automatically constructed according to the foreign keys among relations in $\mathcal{R}_F$. Meanwhile, $\mathcal{R}_F$ is determined by $\mathcal{A}_F$ and $\sigma_F$. $\mathcal{R}_F$ is the union set of relations which contains at least one attribute of $\mathcal{A}_F$ or $\sigma_F$. Hence, the components of query form $F$ are actually determined by $\mathcal{A}_F$ and $\sigma_F$. As we mentioned, only $\mathcal{A}_F$ and $\sigma_F$ are visible to the user in the user interface. In this paper, we focus on the projection and selection components of a query form. Ad-hoc join is not handled by our dynamic query form because join is not a part of the query form and is invisible for users. As for "Aggregation" and "Order by" in SQL, there are limited options for users. For example, "Aggregation" can only be MAX, MIN, AVG, and so on; and "Order by" can only be "increasing order" and "decreasing order". Our dynamic query form can be easily extended to include those options by implementing them as dropdown boxes in the user interface of the query form.

### 3.2 Query Results

To decide whether a query form is desired or not, a user does not have time to go over every data instance in the query results. In addition, many database queries output a huge amount of data instances. In order to avoid this "Many-Answer" problem [10], we only output a compressed result table to show a high-level view of the query results first. Each instance in the compressed table represents a cluster of actual data instances. Then, the user can click through interested clusters to view the detailed data instances. Figure 2 shows the flow of user actions. The compressed high-level view of query results is proposed in [24]. There are many one-pass clustering algorithms for generating the compressed view efficiently [34], [5]. In our implementation, we choose the incremental data clustering framework [5] because of the efficiency issue. Certainly, different data clustering methods would have different compressed views for the users. Also, different clustering methods are preferable to different data types. In this paper, clustering is just to provide a better view of the query results for the user. The system developers can select a different clustering algorithm if needed.

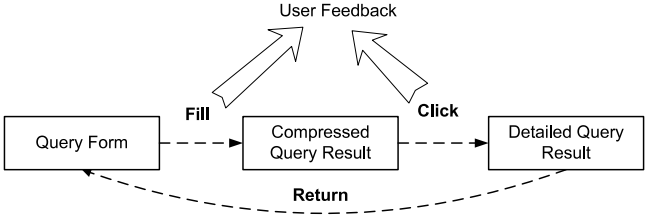



Fig. 2. User Actions

Another important usage of the compressed view is to collect the user feedback. Using the collected feed-

back, we can estimate the goodness of a query form so that we could recommend appropriate query form components. In real world, end-users are reluctant to provide explicit feedback [19]. The click-through on the compressed view table is an implicit feedback to tell our system which cluster (or subset) of data instances is desired by the user. The clicked subset is denoted by $D_{uf}$. Note that $D_{uf}$ is only a subset of all user desired data instances in the database. But it can help our system generate recommended form components that help users discover more desired data instances. In some recommendation systems and search engines, the end-users are also allowed to provide the negative feedback. The negative feedback is a collection of the data instances that are not desired by the users. In the query form results, we assume most of the queried data instances are not desired by the users because if they are already desired, then the query form generation is almost done. Therefore, the positive feedback is more informative than the negative feedback in the query form generation. Our proposed model can be easily extended for incorporating the negative feedback.

## 4 Ranking Metric

Query forms are designed to return the user's desired result. There are two traditional measures to evaluate the quality of the query results: *precision* and *recall* [30]. Query forms are able to produce different queries by different inputs, and different queries can output different query results and achieve different *precisions* and *recalls*, so we use *expected precision* and *expected recall* to evaluate the expected performance of the query form. Intuitively, *expected precision* is the expected proportion of the query results which are interested by the current user. *Expected recall* is the expected proportion of user interested data instances which are returned by the current query form. The user interest is estimated based on the user's click-through on query results displayed by the query form. For example, if some data instances are clicked by the user, these data instances must have high user interests. Then, the query form components which can capture these data instances should be ranked higher than other components. Next we introduce some notations and then define expected precision and recall.

**Notations:** Table 2 lists the symbols used in this paper. Let $F$ be a query form with selection condition $\sigma_F$ and projection attribute set $A_F$. Let $D$ be the collection of instances in $\bowtie(\mathcal{R}_F)$. $N$ is the number of data instances in $D$. Let $d$ be an instance in $D$ with a set of attributes $\mathcal{A} = \{A_1, A_2, ..., A_n\}$, where $n = |\mathcal{A}|$. We use $d_{\mathcal{A}_F}$ to denote the projection of instance $d$ on attribute set $\mathcal{A}_F$ and we call it a projected instance. $P(d)$ is the occurrence probability of $d$ in $D$. $P(\sigma_F|d)$ is the probability of $d$ satisfies $\sigma_F$. $P(\sigma_F|d) \in \{0, 1\}$.

TABLE 2
Symbols and Notations

| | |
|---|---|
| $F$ | query form |
| $\mathcal{R}_F$ | set of relations involved in $F$ |
| $\mathcal{A}$ | set of all attributes in $\bowtie(\mathcal{R}_F)$ |
| $\mathcal{A}_F$ | set of projection attributes of query form $F$ |
| $\mathcal{A}_r(F)$ | set of relevant attributes of query form $F$ |
| $\sigma_F$ | set of selection expressions of query form $F$ |
| $\mathcal{OP}$ | set of relational operators in selection |
| $d$ | data instance in $\bowtie(\mathcal{R}_F)$ |
| $D$ | the collection of data instances in $\bowtie(\mathcal{R}_F)$ |
| $N$ | number of data instances in $D$ |
| $d_{\mathcal{A}_1}$ | data instance $d$ projected on attribute set $\mathcal{A}_1$ |
| $D_{\mathcal{A}_1}$ | set of unique values $D$ projected on attribute set $\mathcal{A}_1$ |
| $Q$ | database query |
| $D_Q$ | results of $Q$ |
| $D_{uf}$ | user feedback as clicked instances in $D_Q$ |
| $\alpha$ | fraction of instances desired by users |

$P(\sigma_F|d) = 1$ if $d$ is returned by $F$ and $P(\sigma_F|d) = 0$ otherwise.

Since query form $F$ projects instances to attribute set $A_F$, we have $D_{A_F}$ as a projected database and $P(d_{A_F})$ as the probability of projected instance $d_{A_F}$ in the projected database. Since there are often duplicated projected instances, $P(d_{A_F})$ may be greater than $1/N$. Let $P_u(d)$ be the probability of $d$ being desired by the user and $P_u(d_{A_F})$ be the probability of the user being interested in a projected instance. We give an example below to illustrate those notations.

TABLE 3
Data Table

| ID | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ |
|---|---|---|---|---|---|
| $I_1$ | $a_1$ | $b_1$ | $c_1$ | 20 | 1 |
| $I_2$ | $a_1$ | $b_2$ | $c_2$ | 20 | 100 |
| $I_3$ | $a_1$ | $b_2$ | $c_3$ | 30 | 99 |
| $I_4$ | $a_1$ | $b_1$ | $c_4$ | 20 | 1 |
| $I_5$ | $a_1$ | $b_3$ | $c_4$ | 10 | 2 |

*Example 1:* Consider a query form $F_i$ with one relational data table shown in Table 3. There are 5 data instances in this table, $D = \{I_1, I_2, ..., I_5\}$, with 5 data attributes $\mathcal{A} = \{C_1, C_2, C_3, C_4, C_5\}$, $N = 5$. Query form $F_i$ executes a query $Q$ as "`SELECT` $C_2$, $C_5$ `FROM` $D$ `WHERE` $C_2 = b_1$ `OR` $C_2 = b_2$". The query result is $D_Q = \{I_1, I_2, I_3, I_4\}$ with projected on $C_2$ and $C_5$. Thus $P(\sigma_{F_i}|d)$ is 1 for $I_1$ to $I_4$ and is zero for $I_5$. Instance $I_1$ and $I_4$ have the same projected values so we can use $I_1$ to represent both of them and $P(I_{1_{C_2,C_5}}) = 2/5$.

**Metrics:** We now describe the two measures *expected precision* and *expected recall* for query forms.

*Definition 2:* Given a set of projection attributes $\mathcal{A}$ and a universe of selection expressions $\sigma$, the *expected precision* and *expected recall* of a query form $F$=($\mathcal{A}_F$, $\mathcal{R}_F$, $\sigma_F$, $\bowtie(\mathcal{R}_F)$) are $Precision_E(F)$ and $Recall_E(F)$

respectively, i.e.,

$$Precision_E(F) = \frac{\sum_{d \in D_{\mathcal{A}_F}} P_u(d_{\mathcal{A}_F}) P(d_{\mathcal{A}_F}) P(\sigma_F|d) N}{\sum_{d \in D_{\mathcal{A}_F}} P(d_{\mathcal{A}_F}) P(\sigma_F|d) N}, \quad (1)$$

$$Recall_E(F) = \frac{\sum_{d \in D_{\mathcal{A}_F}} P_u(d_{\mathcal{A}_F}) P(d_{\mathcal{A}_F}) P(\sigma_F|d) N}{\alpha N}, \quad (2)$$

where $\mathcal{A}_F \subseteq \mathcal{A}$, $\sigma_F \in \sigma$, and $\alpha$ is the fraction of instances desired by the user, i.e., $\alpha = \sum_{d \in D} P_u(d) P(d)$.

The numerators of both equations represent the expected number of data instances in the query result that are desired by the user. In the query result, each data instance is projected to attributes in $\mathcal{A}_F$. So $P_u(d_{\mathcal{A}_F})$ represents the user interest on instance $d$ in the query result. $P(d_{\mathcal{A}_F})N$ is the expected number of rows in $D$ that the projected instance $d_{\mathcal{A}_F}$ represents. Further, given a data instance $d \in D$, $d$ being desired by the user and $d$ satisfying $\sigma_F$ are independent. Therefore, the product of $P_u(d_{\mathcal{A}_F})$ and $P(\sigma_F|d)$ can be interpreted as the probability of $d$ being desired by the user and meanwhile $d$ being returned in the query result. Summing up over all data instances gives the expected number of data instance in the query result being desired by the user.

Similarly, the denominator of Eq.(1) is simply the number of instances in the query result. The denominator of Eq.(2) is the expected number of instances desired by the user in the whole database. In both equations $N$ cancels out so we do not need to consider $N$ when estimating precision and recall. The probabilities in these equations can be estimated using methods described in Section 5. $\alpha = \sum_{d \in D} P_u(d) P(d)$ is the fraction of instances desired by user. $P(d)$ is given by $D$. $P_u(d)$ could be estimated by the method described in Section 5.1.

For example, suppose in Example 1, after projecting on $C_2, C_5$, there are only 4 distinct instances $I_1$, $I_2$, $I_3$, and $I_5$ ($I_4$ has the same projected values as $I_1$). The probability of these projected instances are 0.4, 0.2, 0.2, and 0.2, respectively. Suppose $P_u$ for $I_2$ and $I_3$ are 0.9 and $P_u$ for $I_1$ and $I_5$ are 0.03. The expected precision equals $\frac{0.03\times0.4+0.9\times0.2+0.9\times0.2+0}{0.4+0.2+0.2+0} = 0.465$. Suppose $\alpha = 0.4$, then the expected recall equals $(0.03 \times 0.4 + 0.9 \times 0.2 + 0.9 \times 0.2 + 0)/0.4 = 0.93$.

Considering both *expected precision* and *expected recall*, we derive the overall performance measure, *expected F-Measure* as shown in Equation 3. Note that $\beta$ is a constant parameter to control the preference on *expected precision* or *expected recall*.

*Definition 3:* Given a set of projection attributes $\mathcal{A}$ and an universe of selection expressions $\sigma$, the *expected F-Measure* of a query form $F$=($\mathcal{A}_F$, $\mathcal{R}_F$, $\sigma_F$, $\bowtie(\mathcal{R}_F)$) is $FScore_E(F)$, i.e.,

$$FScore_E(F) = \frac{(1+\beta^2) \cdot Precision_E(F) \cdot Recall_E(F)}{\beta^2 \cdot Precision_E(F) + Recall_E(F)}.$$

**Problem Definition:** In our system, we provide a ranked list of query form components for the user. Problem 1 is the formal statement of the ranking problem.

*Problem 1:* Let the current query form be $F_i$ and the next query form be $F_{i+1}$, construct a ranking of all candidate form components, in descending order of $FScore_E(F_{i+1})$, where $F_{i+1}$ is the query form of $F_i$ enriched by the corresponding form component.

$FScore_E(F_{i+1})$ is the estimated goodness of the next query form $F_{i+1}$. Since we aim to maximize the goodness of the next query form, the form components are ranked in descending order of $FScore_E(F_{i+1})$. In the next section, we will discuss how to compute the $FScore_E(F_{i+1})$ for a specific form component.

## 5 Estimation of Ranking Score

### 5.1 Ranking Projection Form Components

DQF provides a two-level ranked list for projection components. The first level is the ranked list of entities. The second level is the ranked list of attributes in the same entity. We first describe how to rank each entity's attributes locally, and then describe how to rank entities.

#### 5.1.1 Ranking Attributes

Suggesting projection components is actually suggesting attributes for projection. Let the current query form be $F_i$, the next query form be $F_{i+1}$. Let $\mathcal{A}_{F_i} = \{A_1, A_2, ..., A_j\}$, and $\mathcal{A}_{F_{i+1}} = \mathcal{A}_{F_i} \cup \{A_{j+1}\}$, $j+1 \leq |\mathcal{A}|$. $A_{j+1}$ is the projection attribute we want to suggest for the $F_{i+1}$, which maximizes $FScore_E(F_{i+1})$. From the Definition 3, we obtain $FScore_E(F_{i+1})$ as follows:

$$\begin{aligned} &FScore_E(F_{i+1}) \\ =&(1+\beta^2) \cdot \frac{Precision_E(F_{i+1}) \cdot Recall_E(F_{i+1})}{\beta^2 \cdot Precision_E(F_{i+1}) + Recall_E(F_{i+1})} \\ =&\frac{(1+\beta^2) \cdot \sum_{d \in D_{\mathcal{A}_{F_{i+1}}}} P_u(d_{\mathcal{A}_{F_{i+1}}}) P(d_{\mathcal{A}_{F_{i+1}}}) P(\sigma_{F_{i+1}}|d)}{\sum_{d \in D} P(d_{\mathcal{A}_{F_{i+1}}}) P(\sigma_{F_{i+1}}|d) + \beta^2 \alpha}. \end{aligned} \quad (3)$$

Note that adding a projection component $A_{j+1}$ does not affect the selection part of $F_i$. Hence, $\sigma_{F_{i+1}} = \sigma_{F_i}$ and $P(\sigma_{F_{i+1}}|d) = P(\sigma_{F_i}|d)$. Since $F_i$ is already used by the user, we can estimate $P(d_{\mathcal{A}_{F_{i+1}}}) P(\sigma_{F_{i+1}}|d)$ as follows. For each query submitted for form $F_i$, we keep the query results including all columns in $\mathcal{R}_F$. Clearly, for those instances not in query results their $P(\sigma_{F_{i+1}}|d) = 0$ and we do not need to consider them. For each instance $d$ in the query results, we simply count the number of times they appear in the results and $P(d_{\mathcal{A}_{F_{i+1}}}) P(\sigma_{F_{i+1}}|d)$ equals the occurrence count divided by $N$.

Now we only need to estimate $P_u(d_{\mathcal{A}_{F_{i+1}}})$. As for the projection components, we have:

$$\begin{aligned} P_u(d_{\mathcal{A}_{F_{i+1}}}) &= P_u(d_{A_1}, ..., d_{A_j}, d_{A_{j+1}}) \\ &= P_u(d_{A_{j+1}}|d_{\mathcal{A}_{F_i}})P_u(d_{\mathcal{A}_{F_i}}). \end{aligned} \quad (4)$$

$P_u(d_{\mathcal{A}_{F_i}})$ in Eq.(4) can be estimated by the user's click-through on results of $F_i$. The click-through $D_{uf} \subseteq D$ is a set of data instances which are clicked by the user in previous query results. We apply *kernel density estimation* method to estimate $P_u(d_{\mathcal{A}_{F_i}})$. Each $d_b \in D_{uf}$ represents a Gaussian distribution of the user's interest. Then,

$$P_u(d_{\mathcal{A}_{F_i}}) = \frac{1}{|D_{uf}|} \sum_{x \in D_{uf}} \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{d(d_{\mathcal{A}_{F_i}}, x_{\mathcal{A}_{F_i}})^2}{2\sigma^2}),$$

where $d(\cdot,\cdot)$ denotes the distance between two data instances, $\sigma^2$ is the variance of Gaussian models. For numerical data, the Euclidean distance is a conventional choice for distance function. For categorical data, such as string, previous literatures propose several context-based similarity functions which can be employed for categorical data instances [14] [8].

$P_u(d_{A_{j+1}}|d_{\mathcal{A}_{F_i}})$ in Eq.(4) is not visible in the runtime data, since $d_{A_{j+1}}$ has not been used before $F_{i+1}$. We can only estimate it from other data sources. We mainly consider the following two data-driven approaches to estimate the conditional probability $P_u(d_{A_{j+1}}|d_{\mathcal{A}_{F_i}})$.

- *Workload-Driven Approach*: The conditional probability of $P_u(d_{A_{j+1}}|d_{\mathcal{A}_{F_i}})$ could be estimated from query results of historic queries. If a lot of users queried attributes $\mathcal{A}_{F_i}$ and $A_{j+1}$ together on instance $d$, then $P_u(d_{A_{j+1}}|d_{\mathcal{A}_{F_i}})$ must be high.
- *Schema-Driven Approach*: The database schema implies the relations of the attributes. If two attributes are contained by the same entity, then they are more relevant.

Each of the two approaches has its own drawback. The workload-driven approach has the **cold-start** problem since it needs a large amount of queries. The schema-driven approach is not able to identify the difference of the same entity's attributes. In our system, we combined the two approaches as follows:

$$\begin{aligned} &P_u(d_{A_{j+1}}|d_{\mathcal{A}_{F_i}}) \\ &= (1-\lambda)P_b(d_{A_{j+1}}|d_{\mathcal{A}_{F_i}}) + \lambda sim(A_{j+1}, \mathcal{A}_{F_i}), \end{aligned}$$

where $P_b(d_{A_{j+1}}|d_{\mathcal{A}_{F_i}})$ is the probability estimated from the historic queries, $sim(A_{j+1}, \mathcal{A}_{F_i})$ is the similarity between $A_{j+1}$ and $\mathcal{A}_{F_i}$ estimated from the database schema, and $\lambda$ is a weight parameter in $[0, 1]$. $\lambda$ is utilized to balance the workload-driven estimation and schema-driven estimation. Note that

$$sim(A_{j+1}, \mathcal{A}_{F_i}) = 1 - \frac{\sum_{A \in \mathcal{A}_{F_i}} d(A_{j+1}, A)}{|\mathcal{A}_{F_i}| \cdot d_{max}},$$

where $d(A_{j+1}, A)$ is the *schema distance* between the attribute $A_{j+1}$ and $A$ in the schema graph, $d_{max}$ is the diameter of the schema graph. The idea of considering a database schema as a graph is initially proposed by [16]. They proposed a PageRank-like algorithm to compute the importance of an attribute in the schema according to the schema graph. In this paper, we utilize the schema graph to compute the relevance of two attributes. A database schema graph is denoted by $G = (\mathcal{R}, \mathcal{FK}, \xi, \mathcal{A})$, in which $\mathcal{R}$ is the set of nodes representing the relations, $\mathcal{A}$ is the set of attributes, $\mathcal{FK}$ is the set of edges representing the foreign keys, and $\xi : \mathcal{A} \longrightarrow \mathcal{R}$ is an attribute labeling function to indicate which relation contains the attribute. Based on the database schema graph, the *schema distance* is defined as follows.

*Definition 4:* **Schema Distance** Given two attributes $A_1$,$A_2$ with a database schema graph $G$=($\mathcal{R}$,$\mathcal{FK}$,$\xi$,$\mathcal{A}$), $A_1 \in \mathcal{A}$, $A_2 \in \mathcal{A}$, the schema distance between $A_1$ and $A_2$ is $d(A_1, A_2)$, which is the length of the shortest path between node $\xi(A_1)$ and node $\xi(A_2)$.

#### 5.1.2 Ranking Entities

The ranking score of an entity is just the averaged $FScore_E(F_{i+1})$ of that entity's attributes. Intuitively, if one entity has many high score attributes, then it should have a higher rank.

### 5.2 Ranking Selection Form Components

The selection attributes must be relevant to the current projected entities, otherwise that selection would be meaningless. Therefore, the system should first find out the relevant attributes for creating the selection components. We first describe how to select relevant attributes and then describe a naive method and a more efficient one-query method to rank selection components.

#### 5.2.1 Relevant Attribute Selection

The relevance of attributes in our system is measured based on the *database schema* as follows.

*Definition 5:* **Relevant Attributes** Given a database query form $F$ with a schema graph $G$=($\mathcal{R}$,$\mathcal{FK}$,$\xi$,$\mathcal{A}$), the relevant attributes is: $\mathcal{A}_r(F) = \{A | A \in \mathcal{A}, \exists A_j \in \mathcal{A}_F, d(A, A_j) \leq t\}$, where $t$ is a user-defined threshold and $d(A, A_j)$ is the schema distance defined in Definition 4.

The choice of $t$ depends on how compact of the schema is designed. For instance, some databases put all attributes of one entity into a relation, then $t$ could be 1. Some databases separate all attributes of one entity into several relations, then $t$ could be greater than 1. Using the depth-first traversing of the database schema graph, $\mathcal{A}_r(F)$ can be obtained in $O(|\mathcal{A}_r(F)| \cdot t)$.

#### 5.2.2 Ranking Selection Components

For enriching selection form components of a query form, the set of projection components $\mathcal{A}_F$ is fixed, i.e.,

$\mathcal{A}_{F_{i+1}} = \mathcal{A}_{F_i}$. Therefore, $FScore_E(F_{i+1})$ only depends on $\sigma_{F_{i+1}}$.

For the simplicity of the user interface, most query forms' selection components are simple binary relations in the form of "$A_j$ *op* $c_j$", where $A_j$ is an attribute, $c_j$ is a constant and *op* is a relational operator. The *op* operator could be '=', '$\geq$' '$\leq$' and so on. In each cycle, the system provides a ranked list of such binary relations for users to enrich the selection part. Since the total number of binary relations are so large, we only select the best selection component for each attribute.

For attribute $A_s$, $A_s \in \mathcal{A}_r(F)$, let $\sigma_{F_{i+1}} = \sigma_{F_i} \cup \{s\}$, $s \in \sigma$ and $s$ contains $A_s$. According to the formula of $FScore_E(F_{i+1})$, in order to find the $s \in \sigma$ that maximizes the $FScore_E(F_{i+1})$, we only need to estimate $P(\sigma_{F_{i+1}}|d)$ for each data instance $d \in D$. Note that, in our system, $\sigma_F$ represents a conjunctive expression, which connects all elemental binary expressions by *AND*. $\sigma_{F_{i+1}}$ exists if and only if both $\sigma_{F_i}$ and $s$ exist. Hence, $\sigma_{F_{i+1}} \Leftrightarrow \sigma_{F_i} \wedge s$. Then, we have:

$$P(\sigma_{F_{i+1}}|d) = P(\sigma_{F_i}, s|d) = P(s|\sigma_{F_i}, d)P(\sigma_{F_i}|d). \quad (5)$$

$P(\sigma_{F_i}|d)$ can be estimated by previous queries executed on query form $F_i$, which has been discussed in Section 5.1. $P(s|\sigma_{F_i}, d)$ is 1 if and only if $d$ satisfies $\sigma_{F_i}$ and $s$, otherwise it is 0. The only problem is to determine the space of $s$, since we have to enumerate all the $s$ to compute their scores. Note that $s$ is a binary expression in the form of "$A_s$ $op_s$ $c_s$", in which $A_s$ is fixed and given. $op_s \in \mathcal{OP}$ where $\mathcal{OP}$ is a finite set of relational operators, $\{=, \geq, \leq, ...\}$, and $c_s$ belongs to the data domain of $A_s$ in the database. Therefore, the space of $s$ is a finite set $\mathcal{OP} \times D_{A_s}$. In order to efficiently estimate the new $FScore$ induced by a query condition $s$, we propose the `One-query` method in this paper. The idea of `One-query` is simple: we sort the values of an attribute in $s$ and incrementally compute the $FScore$ on all possible values for that attribute.

To find the best selection component for the next query form, the first step is to query the database to retrieve the data instances. In Section 5.2, Eq. (5) presents $P(\sigma_{F_{i+1}}|d)$ depends on the previous query conditions $\sigma_{F_i}$. If $P(\sigma_{F_i}|d) = 0$, $P(\sigma_{F_{i+1}}|d)$ must be 0. Hence, in order to compute the $P(\sigma_{F_{i+1}}|d)$ for each $d \in D$, we don't need to retrieve all data instances in the database. What we need is only the set of data instances $D' \subseteq D$ such that each $d \in D'$ satisfies $P(\sigma_{F_i}|d) > 0$. So the selection of `One-Query`'s query is the union of query conditions executed in $F_i$.

In addition, `One-Query` algorithm does not send each query condition $s$ to the database engine to select data instances, which would be a heavy burden for the database engine since the number of query conditions is large. Instead, it retrieves the set of data instances $D'$, and checks every data instance with every query condition by its own. For this purpose, the algorithm needs to know the values of all selection attributes of $D'$. Hence, `One-Query` adds all the selection attributes into the projections of the query.

Algorithm 1 describes the algorithm of the `One-Query`'s query construction. The function `GenerateQuery` is to generate the database query based on the given set of projection attributes $\mathcal{A}_{one}$ with selection expression $\sigma_{one}$.

**Algorithm 1:** QueryConstruction

**Data**: $\mathcal{Q} = \{Q_1, Q_2, ..., \}$ is the set of previous queries executed on $F_i$.
**Result**: $Q_{one}$ is the query of `One-Query`
**begin**
  $\sigma_{one} \longleftarrow 0$
  **for** $Q \in \mathcal{Q}$ **do**
    $\sigma_{one} \longleftarrow \sigma_{one} \vee \sigma_Q$
  $\mathcal{A}_{one} \longleftarrow \mathcal{A}_{F_i} \cup \mathcal{A}_r(F_i)$
  $Q_{one} \longleftarrow$ `GenerateQuery`($\mathcal{A}_{one}$,$\sigma_{one}$)

When the system receives the result of the query $Q_{one}$ from the database engine, it calls the second algorithm of `One-Query` to find the best query condition.

We first discuss the "$\leq$" condition. The basic idea of this algorithm is based on a simple property. For a specific attribute $A_s$ with a data instance $d$, given two conditions:

$$s_1 : A_s \leq a_1,$$
$$s_2 : A_s \leq a_2,$$

and $a_1 \leq a_2$, if $s_1$ is satisfied, then $s_2$ must be satisfied. Based on this property, we could incrementally compute the FScore of each query condition by scanning one pass of data instances. There are 2 steps to do this.

1) First, we sort the values of $A_s$ in the order of $a_1 \leq a_2 \leq .... \leq a_m$, where $m$ is the number of $A_s$'s values. Let $D_{a_j}$ denote the set of data instances in which $A_s$'s value is equal to $a_j$.
2) Then, we go through every data instance in the order of $A_s$'s value. Let query condition $s_j$ = "$A_s \leq a_j$" and its corresponding FScore be $fscore_j$. According to Eq. (3), $fscore_j$ can be computed as

$$fscore_j = (1 + \beta^2) \cdot n_j / d_j,$$
$$n_j = \sum_{d \in D_{Q_{one}}} P_u(d_{A_{F_i}}) P(d_{A_{F_i}}) P(\sigma_{F_i}|d) P(s_i|d),$$
$$d_j = \sum_{d \in D_{Q_{one}}} P(d_{A_{F_i}}) P(\sigma_{F_i}|d) P(s_i|d) + \alpha\beta^2.$$

For $j > 1$, $n_j$ and $d_j$ can be calculated incremen-

tally:

$$n_j = n_{j-1} + \sum_{d \in D_{a_j}} P_u(d_{A_{F_i}})P(d_{A_{F_i}})P(\sigma_{F_i}|d)P(s_j|d),$$

$$d_j = d_{j-1} + \sum_{d \in D_{a_j}} P(d_{A_{F_i}})P(\sigma_{F_i}|d)P(s_j|d).$$

Algorithm 2 shows the pseudocode for finding the best "$\leq$" condition.

**Algorithm 2:** FindBestLessEqCondition

**Data**: $\alpha$ is the fraction of instances desired by user, $D_{Q_{one}}$ is the query result of $Q_{one}$, $A_s$ is the selection attribute.
**Result**: $s^*$ is the best query condition of $A_s$.
**begin**
  *// sort by $A_s$ into an ordered set $D_{sorted}$*
  $D_{sorted} \longleftarrow \texttt{Sort}(D_{Q_{one}}, A_s)$
  $s^* \longleftarrow \emptyset$, $fscore^* \longleftarrow 0$
  $n \longleftarrow 0$, $d \longleftarrow \alpha\beta^2$
  **for** $i \leftarrow 1$ **to** $|D_{sorted}|$ **do**
    $d \longleftarrow D_{sorted}[i]$
    $s \longleftarrow$ "$A_s \leq d_{A_s}$"
    *// compute fscore of "$A_s \leq d_{A_s}$"*
    $n \longleftarrow n + P_u(d_{A_{F_i}})P(d_{A_{F_i}})P(\sigma_{F_i}|d)P(s|d)$
    $d \longleftarrow d + P(d_{A_{F_i}})P(\sigma_{F_i}|d)P(s|d)$
    $fscore \longleftarrow (1+\beta^2) \cdot n/d$
    **if** $fscore \geq fscore^*$ **then**
      $s^* \longleftarrow s$
      $fscore^* \longleftarrow fscore$

**Complexity:** As for other query conditions, such as "$=$", "$\geq$", we can also find similar incremental approaches to compute their FScore. They all share the sorting result in the first step. And for the second step, all incremental computations can be merged into one pass of scanning $D_{Q_{one}}$. Therefore, the time complexity of finding the best query condition for an attribute is $O(|D_{Q_{one}}| \cdot |\mathcal{A}_{F_i}|)$. Ranking every attribute's selection component is $O(|D_{Q_{one}}| \cdot |\mathcal{A}_{F_i}| \cdot |\mathcal{A}_r(F_i)|)$.

#### 5.2.3 Diversity of Selection Components

Two selection components may have a lot of overlap (or redundancy). For example, if a user is interested in some customers with age between 30 and 45, then two selection components: "$age > 28$" and "$age > 29$" could get similar $FScores$ and similar sets of data instances. Therefore, there is a redundancy of the two selections. Besides a high precision, we also require the recommended selection components should have a high *diversity*. *diversity* is a recent research topic in recommendation systems and web search engines [6] [28]. However, simultaneously maximizing the precision and the diversity is an NP-Hard problem [6]. It cannot be efficiently implemented in an interactive system. In our dynamic query form system, we observe that most redundant selection components are constructed by the same attribute. Thus, we only recommend the best selection component for each attribute.

## 6 Evaluation

The goal of our evaluation is to verify the following hypotheses:

- H1: Is DQF more usable than existing approaches such as static query form and customized query form?
- H2: Is DQF more effective to rank projection and selection components than the baseline method and the random method?
- H3: Is DQF efficient to rank the recommended query form components in an online user interface?

### 6.1 System Implementation and Experimental Setup

We implemented the dynamic query forms as a web-based system using JDK 1.6 with Java Server Page. The dynamic web interface for the query forms used open-source javascript library jQuery 1.4. We used MySQL 5.1.39 as the database engine. All experiments were run using a machine with Intel Core 2 CPU @2.83GHz, 3.5G main memory, and running on Windows XP SP2. Figure 3 shows a system prototype.

**Data sets:** 3 databases: NBA [1], Green Car [2] and Geobase [3] were used in our experiments. Table 4 shows a general description of those databases.

TABLE 4
Data Description

| Name | #*Relations* | #*Attribute* | #*Instances* |
|---|---|---|---|
| NBA | 10 | 180 | 44,590 |
| Green Car | 1 | 17 | 2,187 |
| Geobase | 9 | 32 | 1,329 |

**Form Generation Approaches:** We compared three approaches to generate query forms:

- DQF: The dynamic query form system proposed in this paper.
- SQF: The static query form generation approach proposed in [18]. It also uses query workload. Queries in the workload are first divided into clusters. Each cluster is converted into a query form.
- CQF: The customized query form generation used by many existing database clients, such as Microsoft Access, EasyQuery, ActiveQueryBuilder.

**User Study Setup:** We conducted a user study to evaluate the usability of our approach. We recruited 20

1. http://www.databasebasketball.com
2. http://www.epa.gov/greenvehicles
3. Geobase is a database of geographic information about the USA, which is used in [16]

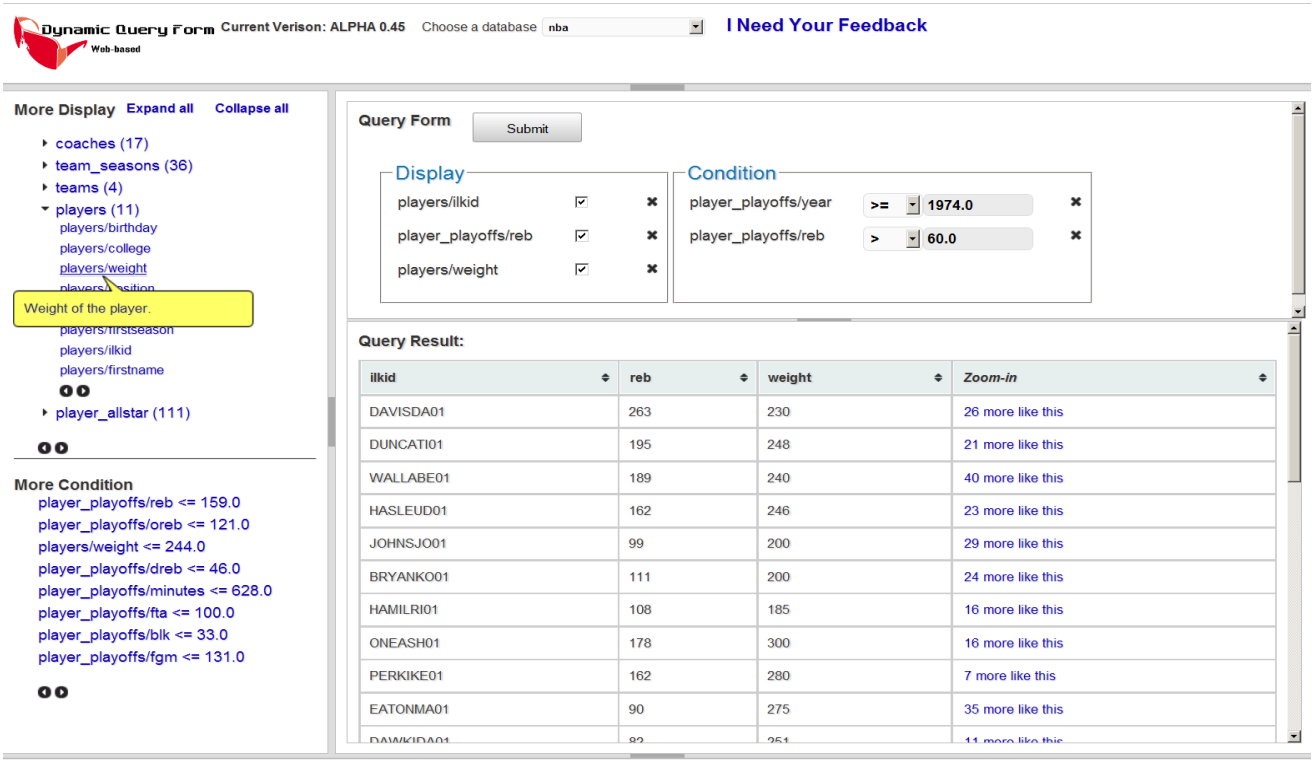



Fig. 3. Screenshot of Web-based Dynamic Query Form

participants of graduate students, UI designers, and software engineers. The system prototype is shown by Figure 3. The user study contains 2 phases, a query collection phase and a testing phase. In the collection phase, each participant used our system to submit some queries and we collected these queries. There were 75 queries collected for `NBA`, 68 queries collected for `Green Car`, and 132 queries for `Geobase`. These queries were used as query workload to train our system (see Section 5.1). In the second phase, we asked each participant to complete 12 tasks (none of these tasks appeared in the workload) listed in Table 5. Each participant used all three form generation approaches to form queries. The order of the three approaches were randomized to remove bias. We set parameter $\lambda = 0.001$ in our experiments because our databases collect a certain amount of historic queries so that we mainly consider the probability estimated from the historic queries.

**Simulation Study Setup:** We also used the collected queries in a larger scale simulation study. We used a cross-validation approach which partitions queries into a training set (used as workload information) and a testing set. We then reported the average performance for testing sets.

### 6.2 User Study Results

**Usability Metrics:** In this paper, we employ some widely used metrics in Human-Computer Interaction and Software Quality for measuring the usability of a system [31], [27]. These metrics are listed in Table 7.

TABLE 7
Usability Metrics

| Metric | Definition |
|---|---|
| $AC_{min}$ | The minimal number of *action* for users |
| $AC$ | The actual number of *action* performed by users |
| $AC_{ratio}$ | $AC_{min}/AC \times 100.0\%$ |
| $FN_{max}$ | The total number of provided UI *function* for users to choose |
| $FN$ | The number of actual used UI *function* by the user |
| $FN_{ratio}$ | $FN/FN_{max} \times 100\%$ |
| $Success$ | The percentage of users successfully completed a specific task |

In database query forms, one *action* means a mouse click or a keyboard input for a textbox. $AC_{min}$ is the minimal number of *actions* for a querying task. One *function* means a provided option for the user to use, such as a query form or a form component. In a web page based system, $FN_{max}$ is the total number of UI components in web pages explored by the user. In this user study, each page at most contains 5 UI components. The smaller $AC_{min}$, $AC$, $FN_{max}$, and $FN$, the better the usability. Similarly, the higher the $AC_{ratio}$, $FN_{ratio}$, and $Success$, the better the usability.

There is a trade-off between $AC_{min}$ and $FN_{max}$. An extreme case is that, we generate all possible query forms in one web page, the user only needs to choose one query form to finish his(or her) query task, so $AC_{min}$ is 1. However, $FN_{max}$ would be the number

TABLE 5
Query Tasks

| Task | SQL | Meaning |
|---|---|---|
| T1 | SELECT ilkid, firstname, lastname FROM players | Find all NBA players' ID and full names. |
| T2 | SELECT p.ilkid, p.firstname, p.lastname FROM players p, player_playoffs_career c WHERE p.ilkid = c.ilkid AND c.minutes > 5000 | Find players who have played more than 5000 minutes in the playoff. |
| T4 | SELECT t.team, t.location, c.firstname, c.lastname, c.year FROM teams t, coaches c WHERE t.team=c.team AND t.location = 'Los Angeles' | Find the name of teams located in Los Angeles with their coaches. |
| T5 | SELECT Models, Hwy_MPG FROM cars WHERE City_MPG > 20 | Find the high way MPG of cars whose city road MPG is greater than 20. |
| T6 | SELECT Models, Displ, Fuel FROM cars WHERE Sales_Area = 'CA' | Find the model, displacement and fuel of cars which is sold in California. |
| T7 | SELECT Models, Displ FROM cars WHERE Veh_Class = 'SUV' | Find the displacement of all SUV cars. |
| T8 | SELECT Models FROM cars WHERE Drive = '4WD' | Find all 4 wheel-driven cars. |
| T9 | SELECT t0.population FROM city t0 WHERE t0.state = 'california' | Find all cities in California with the population of each city. |
| T10 | SELECT t1.state, t0.area FROM state t0, border t1 WHERE t1.name = 'wisconsin' and t0.area > '80000' and t0.name = t1.name | Find the neighbor states of Wisconsin whose area is greater than 80,000 square miles. |
| T11 | SELECT t0.name, t0.length FROM river t0 WHERE t0.state = 'illinois' | Find all rivers across Illinois with each river's length. |
| T12 | SELECT t0.name, t0.elevation, t0.state FROM mountain t0 WHERE t0.elevation > '5000' | Find all mountains whose elevation is greater than 5000 meters and each mountain's state. |

of all possible query forms with their components, which is a huge number. On the other hand, if users have to interact a lot with a system, that system would know better about the user's desire. In that case, the system would cut down many unnecessary functions, so that $FN_{max}$ could be smaller. But $AC_{min}$ would be higher since there are a lot of user interactions.

**User Study Analysis:** Table 6 shows the average result of the usability experiments for those query tasks. As for SQF, we generated 10 static query forms based on the collected user queries for each database (i.e., 10 clusters were generated on the query workload).

The results show that users did not accomplish querying tasks by SQF. The reason is that, SQF is built from the query workload and may not be able to answer ad hoc queries in the query tasks. E.g., SQF does not contain any relevant attributes for query task T3 and T11, so users failed to accomplish the queries by SQF.

TABLE 8
Statistical Test on $FN_{max}$ ( with CQF)

| Task | T1 | T4 | T5 | T10 | T11 | T12 |
|---|---|---|---|---|---|---|
| P Value | 0.0106 | <0.0001 | <0.0001 | 0.0132 | <0.0001 | <0.0001 |

TABLE 9
Statistical Test on $AC$ (with CQF)

| Task | T2 | T3 | T4 | T6 | T8 |
|---|---|---|---|---|---|
| P Value | 0.0199 | 0.0012 | 0.0190 | 0.0199 | 0.0179 |

DQF and CQF are capable of assisting users finish all querying tasks. In 11 of those 12 tasks, DQF has a smaller $FN_{max}$ than CQF. We conduct statistical tests (t-tests) on those 11 tasks with $\alpha$=0.05, and find that 6 of them are statistically significant. Table 8 shows those 6 tasks with their P values. As for $AC$, DQF's average values are smaller than CQF's in all 12 tasks, and 5 tasks have statistically significant difference ($\alpha$=0.05). Table 9 shows the 5 tasks with their P values. The reason why DQF outperforms CQF in some tasks is that, CQF does not provide any intelligent assistance for users to create their query forms. For each form component, the user has to enumerate almost all the entities and attributes to find an appropriate form component for the query form. On the contrary, DQF computes ranked lists of potential query form components at each iteration to assist the user. In those tasks, the user found desired entities and attributes at the top of those ranked lists. Therefore, DQF cut down many unnecessary functions shown to the user.

Overall, from the usability aspect, SQF requires the minimal user actions but may not satisfy ad-hoc user queries. It also tends to generate forms with functions not used by the current user. CQF is capable of satisfying ad-hoc query, but it is difficult for users to explore the entire database and search appropriate form components.

### 6.3 Static vs. Dynamic Query Forms

If a query task is covered by one historical queries in history, then SQF built on those historical queries can satisfy that query task. But the costs of using SQF and DQF to accomplish that task are different. *Form-Complexity* was proposed in [18] to evaluate the cost of using a query form. It is the sum of the number of selection components, projection components, and relations, as shown below:

$$Form - Complexity(F) = |\mathcal{A}_F| + |\sigma_F| + |\mathcal{R}_F|.$$

TABLE 6
Usability Results

| Task | Query Form | $AC_{min}$ | $AC$ | $AC_{ratio}$ | $FN_{max}$ | $FN$ | $FN_{ratio}$ | $Success$ |
|---|---|---|---|---|---|---|---|---|
| T1 | DQF | 6 | 6.7 | 90.0% | 40.0 | 3 | 7.5% | 100.0% |
| | CQF | 6 | 7.0 | 85.7% | 60.0 | 3 | 5% | 100.0% |
| | SQF | 1 | 1.0 | 100.0% | 35.0 | 3 | 8.6% | 44.4% |
| T2 | DQF | 7 | 7.7 | 91.0% | 65.0 | 4 | 6.2% | 100.0% |
| | CQF | 8 | 10.0 | 80.0% | 86.7 | 4 | 4.6% | 100.0% |
| | SQF | 1 | 1.0 | 100.0% | 38.3 | 4 | 10.4% | 16.7% |
| T3 | DQF | 10 | 10.7 | 93.5% | 133.3 | 6 | 3.8% | 100.0% |
| | CQF | 12 | 13.3 | 90.2% | 121.7 | 6 | 4.9% | 100.0% |
| | SQF | 1 | N/A | N/A | N/A | 6 | N/A | 0.0% |
| T4 | DQF | 11 | 11.7 | 94.0% | 71.7 | 6 | 8.4% | 100.0% |
| | CQF | 12 | 13.3 | 90.2% | 103.3 | 6 | 5.8% | 100.0% |
| | SQF | 1 | 1.0 | 100.0% | 70.0 | 6 | 8.6% | 16.7% |
| T5 | DQF | 5 | 5.7 | 87.7% | 28.3 | 3 | 10.6% | 100.0% |
| | CQF | 6 | 6.7 | 90.0% | 56.7 | 3 | 5.3% | 100.0% |
| | SQF | 1 | 1.0 | 100.0% | 10 | 3 | 30.0% | 66.7% |
| T6 | DQF | 7 | 7.7 | 91.0% | 61.7 | 4 | 6.5% | 100.0% |
| | CQF | 8 | 10 | 80.0% | 61.7 | 4 | 6.5% | 100.0% |
| | SQF | 1 | 1 | 100.0% | 23.3 | 4 | 17.2% | 41.7% |
| T7 | DQF | 5 | 6.0 | 83.3% | 48.3 | 3 | 6.2% | 100.0% |
| | CQF | 6 | 6.7 | 90.0% | 50.0 | 3 | 6.0% | 100.0% |
| | SQF | 1 | 1.0 | 100.0% | 18.3 | 3 | 16.4% | 44.4% |
| T8 | DQF | 3 | 3.3 | 91.0% | 21.7 | 2 | 9.2% | 100.0% |
| | CQF | 4 | 4.7 | 85.1% | 26.7 | 2 | 7.5% | 100.0% |
| | SQF | 1 | 1.0 | 100.0% | 38.3 | 2 | 5.2% | 66.7% |
| T9 | DQF | 5 | 6.3 | 79.3% | 31.7 | 3 | 9.5% | 100.0% |
| | CQF | 6 | 6.7 | 90.0% | 36.7 | 3 | 8.2% | 100.0% |
| | SQF | 1 | 1.0 | 100.0% | 106.7 | 3 | 2.8% | 66.7% |
| T10 | DQF | 6 | 6.7 | 90.0% | 43.3 | 4 | 9.2% | 100.0% |
| | CQF | 8 | 8.7 | 92.0% | 63.3 | 4 | 6.3% | 100.0% |
| | SQF | 1 | 1.0 | 100.0% | 75.0 | 4 | 5.3% | 33.3% |
| T11 | DQF | 5 | 6.3 | 79.4% | 36.7 | 3 | 8.2% | 100.0% |
| | CQF | 6 | 6.7 | 90.0% | 50.0 | 3 | 6.0% | 100.0% |
| | SQF | 1 | N/A | N/A | N/A | 3 | N/A | 0.0% |
| T12 | DQF | 7 | 7.7 | 91.0% | 46.7 | 4 | 8.6% | 100.0% |
| | CQF | 8 | 10.0 | 80.0% | 85.0 | 4 | 4.7% | 100.0% |
| | SQF | 1 | 1.0 | 100.0% | 31.7 | 4 | 12.6% | 25.0% |

On the premise of satisfying all users' queries, the complexities of query forms should be as small as possible. DQF generates one customized query form for each query. The average form complexity is 8.1 for NBA, 4.5 for Green Car and 6.7 for Geobase. But for SQF, the complexity is 30 for NBA and 16 for Green Car (2 static query forms). This result shows that, in order to satisfy various query tasks, the statically generated query form has to be more complex.

### 6.4 Effectiveness

We compare the ranking function of DQF with two other ranking methods: the baseline method and the random method. The baseline method ranks projection and selection attributes in ascending order of their schema distance (see Definition 4) to the current query form. For the query condition, it chooses the most frequent used condition in the training set for that attribute. The random method randomly suggests one query form component. The ground truth of the query form component ranking is obtained from the query workloads and stated in Section 6.1.

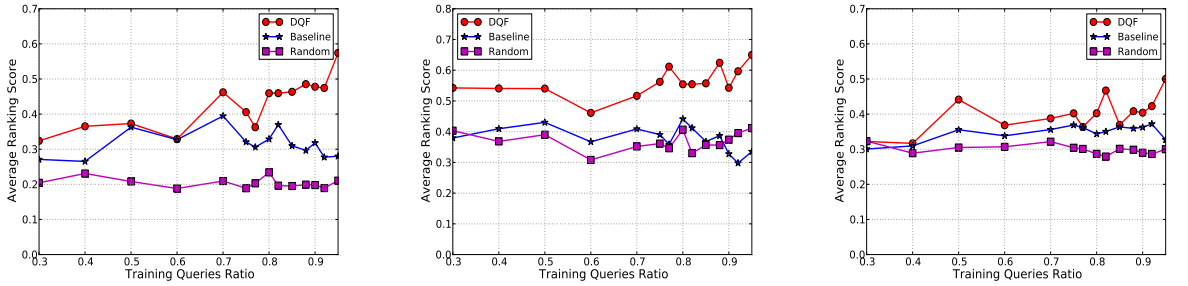
**Ranking projection components:** Ranking score is a supervised method to measure the accuracy of the recommendation. It is obtained by comparing the computed ranking with the optimal ranking. In the optimal ranking, the actual selected component by the user is ranked first. So ranking score evaluates how far away the actual selected component is ranked from the first. The formula of ranks score is computed as follows:

$$Rank_{Score}(Q, A_j) = \frac{1}{\log(\hat{r}(A_j)) + 1},$$

where $Q$ is a test query, $A_j$ is the $j$-th projection attribute of $Q$, $\hat{r}(A_j)$ is the computed rank of $A_j$.

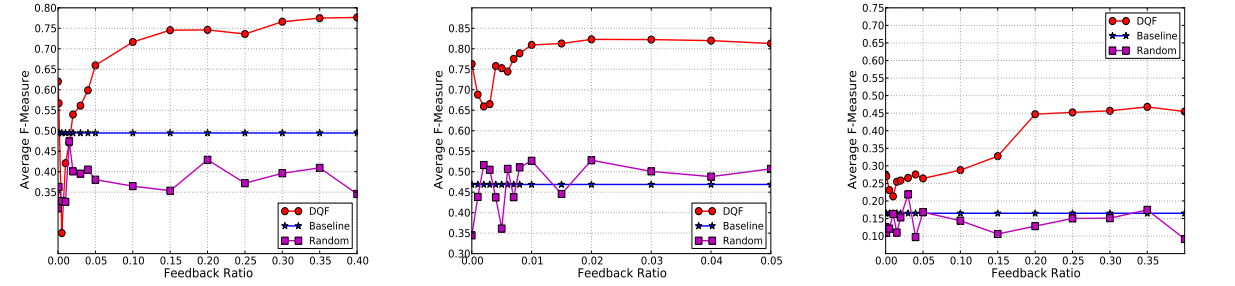
Figure 4 shows the average ranking scores for all queries in the workload. We compare three methods: DQF, Baseline, and Random. The x-axis indicates the portion of the training queries, and the rest queries are used as testing queries. The y-axis indicates the average ranking scores among all the testing queries. DQF always outperforms the baseline method and random method. The gap also grows as the portion of training queries increases because $DQF$ can better utilize the training queries.

**Ranking selection components:** F-Measure is utilized to measure ranking of selection components. Intuitively, if the query result obtained by using the



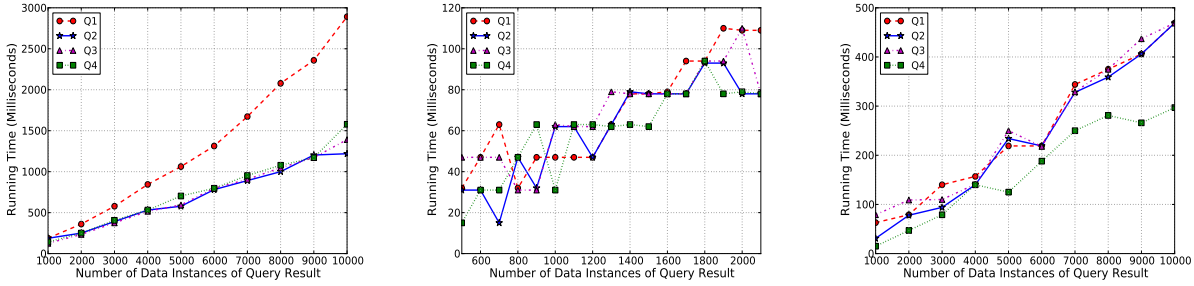

(a) Ranking Scores on NBA Data (b) Ranking Scores on Green Car Data (c) Ranking Scores on Geobase Data

Fig. 4. Ranking Scores of Suggested Selection Components




(a) Average F-Measure for NBA Data (Top 5 Ranked Components) (b) Average F-Measure on Green Car Data (Top 5 Ranked Components) (c) Average F-Measure on Geobase Data (Top 3 Ranked Components)

Fig. 5. Average F-Measure of Suggested Selection Components




(a) Running time on NBA Data (b) Running time on Green Car Data (c) Running time on Geobase Data

Fig. 6. Scalability of Ranking Selection Components

suggested selection component is closer to the actual query result, the F-Measure should be higher. For a test query $Q$, we define the ground truth as the set of data instances returned by the query $Q$. We also constructed a query $\hat{Q}$, where $\hat{Q}$ is identical to $Q$ except for the last selection component. The last selection component of $\hat{Q}$ is constructed by the top ranked component returned by one of the three ranking methods. We then compared the results of $\hat{Q}$ to the ground truth to compute F-Measure. We randomly selected half of queries in the workload as training set and the rest as testing. Since `DQF` uses user's click-through as implicit feedback, we randomly selected some small portion (`Feedback Ratio`) of the ground truth as click-through.

Figure 5 shows the F-Measure ($\beta$=2) of all methods on the data sets. The x-axis of those figures indicates the `Feedback Ratio` over the whole ground truth. The y-axis is the average F-Measure value among all collected queries. From those figures, `DQF` even performs well when there is no click-through information ( `Feedback Ratio`=0).

### 6.5 Efficiency

The run-time cost of ranking projection and selection components for `DQF` depends on the current form components and the query result size. Thus we selected 4 complex queries with large result size for each data set. Table 10, Table 11 and Table 12 list these queries, where those join conditions are implicit inner joins and written in WHERE clause. We varied the query result size by query paging in MySQL engine. The running times of ranking projection are

all less than 1 millisecond, since DQF only computes the schema distance and conditional probabilities of attributes. Figure 6 shows the time for DQF to rank selection components for queries on the data sets. The results show that the execution time grows approximately linearly with respect to the query result size. The execution time is between 1 to 3 seconds for NBA when the results contain 10000 records, less than 0.11 second for Green Car when the results contain 2000 records, and less than 0.5 second for Geobase when results contain 10000 records. So DQF can be used in an interactive environment.

TABLE 10
NBA's Queries in Scalability Test

| Query | SQL |
|---|---|
| $Q1$ | SELECT t0.coachid, t2.leag, t2.location, t2.team, t3.d_blk, t1.fta, t0.season_win, t1.fgm FROM coaches t0, player_regular_season t1, teams t2, team_seasons t3 WHERE t0.team = t1.team and t1.team = t3.team and t3.team = t2.team |
| $Q2$ | SELECT t2.lastname, t2.firstname, t1.won FROM player_regular_season t0, team_seasons t1, players t2 WHERE t1.team = t0.team and t0.ilkid = t2.ilkid |
| $Q3$ | SELECT t0.lastname, t0.firstname FROM players t0, player_regular_season t1, team_seasons t2 WHERE t2.team = t1.team and t1.ilkid = t0.ilkid |
| $Q4$ | SELECT t0.won, t3.name, t2.h_feet FROM team_seasons t0, player_regular_season t1, players t2, teams t3 WHERE t3.team = t0.team and t0.team = t1.team and t1.ilkid = t2.ilkid |

TABLE 11
Green Car's Queries in Scalability Test

| Query | SQL |
|---|---|
| $Q1$ | SELECT Underhood_ID, Displ, Hwy_MPG FROM cars WHERE City_MPG <= '51.0' |
| $Q2$ | SELECT Model FROM cars WHERE Cyl <= '12.0' |
| $Q3$ | SELECT Model, Underhood_ID, Trans FROM cars WHERE City_MPG <= '30.0' and Cmb_MPG <= '34.0' |
| $Q4$ | SELECT Model FROM cars |

## 7 Conclusion and Future Work

In this paper we propose a dynamic query form generation approach which helps users dynamically generate query forms. The key idea is to use a probabilistic model to rank form components based on user preferences. We capture user preference using both historical queries and run-time feedback such as click-through. Experimental results show that the dynamic approach often leads to higher success rate and simpler query forms compared with a static approach. The ranking of form components also makes it easier for users to customize query forms. As future work, we will study how our approach can be extended to non relational data.

TABLE 12
Geobase's Queries in Efficiency Test

| Query | SQL |
|---|---|
| $Q1$ | SELECT t2.name, t0.name, t1.elevation, t4.name, t5.name, t3.name, t3.area, t4.population FROM road t0, mountain t1, highest_point t2, lake t3, state t4, border t5 WHERE t1.elevation = '6194.0' and t3.area = '82362.0' and t2.elevation = '6194.0' and t4.name = t2.state and t2.state = t1.state |
| $Q2$ | SELECT t3.name, t5.name, t2.elevation, t1.name, t4.name, t0.name FROM lake t0, state t1, mountain t2, highest_point t3, border t4, road t5 WHERE t2.elevation = '6194.0' and t0.area = '82362.0' and t1.name = t3.state and t3.state = t2.state |
| $Q3$ | SELECT t4.name, t2.name, t3.elevation, t0.name, t1.name, t5.name FROM state t0, border t1, road t2, mountain t3, highest_point t4, lake t5 WHERE t3.elevation = '6194.0' and t0.name = t4.state and t4.state = t3.state |
| $Q4$ | SELECT t0.elevation, t1.population, t2.state FROM mountain t0, state t1, road t2 WHERE t1.population < '2.367E7' and t0.state = t1.name |

As for the future work, we plan to develop multiple methods to capture the user's interest for the queries besides the click feedback. For instance, we can add a text-box for users to input some keywords queries. The relevance score between the keywords and the query form [12] can be incorporated into the ranking of form components at each step.

## Acknowledgement

The work is partially supported by NSF grants IIS-0546280, HRD-0833093, and CNS-1126619.

## References

[1] Cold Fusion. http://www.adobe.com/products/coldfusion/.
[2] DBPedia. http://DBPedia.org.
[3] EasyQuery. http://devtools.korzh.com/eq/dotnet/.
[4] Freebase. http://www.freebase.com.
[5] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. In *Proceedings of VLDB*, pages 81–92, Berlin, Germany, September 2003.
[6] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying search results. In *Proceedings of WSDM*, pages 5–14, Barcelona, Spain, February 2009.
[7] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In *CIDR*, 2003.
[8] S. Boriah, V. Chandola, and V. Kumar. Similarity measures for categorical data: A comparative evaluation. In *Proceedings of SIAM International Conference on Data Mining (SDM 2008)*, pages 243–254, Atlanta, Georgia, USA, April 2008.
[9] G. Chatzopoulou, M. Eirinaki, and N. Polyzotis. Query recommendations for interactive database exploration. In *Proceedings of SSDBM*, pages 3–18, New Orleans, LA, USA, June 2009.
[10] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic information retrieval approach for ranking of database query results. *ACM Trans. Database Syst. (TODS)*, 31(3):1134–1168, 2006.
[11] K. Chen, H. Chen, N. Conway, J. M. Hellerstein, and T. S. Parikh. Usher: Improving data quality with dynamic forms. In *Proceedings of ICDE conference*, pages 321–332, Long Beach, California, USA, March 2010.

[12] E. Chu, A. Baid, X. Chai, A. Doan, and J. F. Naughton. Combining keyword search and forms for ad hoc querying of databases. In *Proceedings of ACM SIGMOD Conference*, pages 349–360, Providence, Rhode Island, USA, June 2009.

[13] S. Cohen-Boulakia, O. Biton, S. Davidson, and C. Froidevaux. Bioguidesrs: querying multiple sources with a user-centric perspective. *Bioinformatics*, 23(10):1301–1303, 2007.

[14] G. Das and H. Mannila. Context-based similarity measures for categorical databases. In *Proceedings of PKDD 2000*, pages 201–210, Lyon, France, September 2000.

[15] W. B. Frakes and R. A. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[16] M. Jayapandian and H. V. Jagadish. Automated creation of a forms-based database query interface. In *Proceedings of the VLDB Endowment*, pages 695–709, August 2008.

[17] M. Jayapandian and H. V. Jagadish. Expressive query specification through form customization. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, pages 416–427, Nantes, France, March 2008.

[18] M. Jayapandian and H. V. Jagadish. Automating the design and construction of query forms. *IEEE TKDE*, 21(10):1389–1402, 2009.

[19] T. Joachims and F. Radlinski. Search engines that learn from implicit feedback. *IEEE Computer (COMPUTER)*, 40(8):34–40, 2007.

[20] N. Khoussainova, M. Balazinska, W. Gatterbauer, Y. Kwon, and D. Suciu. A case for a collaborative query management system. In *Proceedings of CIDR*, Asilomar, CA, USA, January 2009.

[21] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. Snipsuggest: Context-aware autocompletion for sql. *PVLDB*, 4(1):22–33, 2010.

[22] J. C. Kissinger, B. P. Brunk, J. Crabtree, M. J. Fraunholz, B. Gajria, A. J. Milgram, D. S. Pearson, J. Schug, A. Bahl, S. J. Diskin, H. Ginsburg, G. R. Grant, D. Gupta, P. Labo, L. Li, M. D. Mailman, S. K. McWeeney, P. Whetzel, C. J. Stoeckert, and J. . D. S. Roos. The plasmodium genome database: Designing and mining a eukaryotic genomics resource. *Nature*, 419:490–492, 2002.

[23] C. Li, N. Yan, S. B. Roy, L. Lisham, and G. Das. Facetedpedia: dynamic generation of query-dependent faceted interfaces for wikipedia. In *Proceedings of WWW*, pages 651–660, Raleigh, North Carolina, USA, April 2010.

[24] B. Liu and H. V. Jagadish. Using trees to depict a forest. *PVLDB*, 2(1):133–144, 2009.

[25] P. Mork, R. Shaker, A. Halevy, and P. Tarczy-Hornoch. Pql: a declarative query language over dynamic biological schemata. In *In Proceedings of American Medical Informatics Association Fall Symposium*, pages 533–537, San Antonio, Texas, 2007.

[26] A. Nandi and H. V. Jagadish. Assisted querying using instant-response interfaces. In *Proceedings of ACM SIGMOD*, pages 1156–1158, 2007.

[27] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, San Francisco, 1993.

[28] D. Rafiei, K. Bharat, and A. Shukla. Diversifying web search results. In *Proceedings of WWW*, pages 781–790, Raleigh, North Carolina, USA, April 2010.

[29] S. B. Roy, H. Wang, U. Nambiar, G. Das, and M. K. Mohania. Dynacet: Building dynamic faceted search systems over databases. In *Proceedings of ICDE*, pages 1463–1466, Shanghai, China, March 2009.

[30] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1984.

[31] A. Seffah, M. Donyaee, R. B. Kline, and H. K. Padda. Usability measurement and metrics: A consolidated model. *Software Quality Journal*, 14(2):159–178, 2006.

[32] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *Proceedings of SIGMOD*, pages 535–548, Providence, Rhode Island, USA, September 2009.

[33] G. Wolf, H. Khatri, B. Chokshi, J. Fan, Y. Chen, and S. Kambhampati. Query processing over incomplete autonomous databases. In *Proceedings of VLDB*, pages 651–662, 2007.

[34] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. In *Proceedings of SIGMOD*, pages 103–114, Montreal, Canada, June 1996.

[35] S. Zhu, T. Li, Z. Chen, D. Wang, and Y. Gong. Dynamic active probing of helpdesk databases. *Proc. VLDB Endow.*, 1(1):748–760, Aug. 2008.

[36] M. M. Zloof. Query-by-example: the invocation and definition of tables and forms. In *Proceedings of VLDB*, pages 1–14, Framingham, Massachusetts, USA, September 1975.

**Liang Tang** Liang Tang received the master degree in computer science from the Department of Computer Science, Sichuan University, Chengdu, China, in 2009. He is currently a Ph.D. student in the School of Computing and Information Sciences, Florida International University, Miami. His research interests are data mining, computing system management, data mining and machine learning.

**Tao Li** Tao Li received the Ph.D. degree in computer science from the Department of Computer Science, University of Rochester, Rochester, NY, in 2004. He is currently an Associate Professor with the School of Computing and Information Sciences, Florida International University, Miami. His research interests are data mining, computing system management, information retrieval, and machine learning. He is a recipient of NSF CAREER Award and multiple IBM Faculty Research Awards.

**Yexi Jiang** Yexi Jiang received the master degree in computer science from the Department of Computer Science, Sichuan University, Chengdu, China, in 2010. He is currently a Ph.D. student in the School of Computing and Information Sciences, Florida International University, Miami. His research interests are system oriented data mining, intelligent cloud, large scale data mining, database and semantic web.

**Zhiyuan Chen** Zhiyuan Chen is an associate professor at the Department of Information Systems, University of Maryland Baltimore County. He received PhD in Computer Science from Cornell University in 2002, and M.S. and B.S. in computer science from Fudan University, China. His research interests are in database systems and data mining, including privacy preserving data mining, data exploration and navigation, health informatics, data integration, XML, automatic database administration, and database compression.