JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 33, 1375-1396 (2017)
DOI: 10.6688/JISE.2017.33.5.16

Adaptive Query Relaxation and Result Categorization Based on Data Distribution and Query Context*

XIAOYAN ZHANG, XIANGFU MENG, YANHUAN TANG AND CHONGCHUN BI
School of Electronic and Information Engineering
Liaoning Technical University
Huludao, 125105 P.R. China
E-mail: marxi@126.com

To address the empty and/or many answer problem of Web database query, this paper proposes a general framework to enable automatically query relaxation and result categorization. The framework consists of two processing parts. The first is query relaxation. In this part, each specified attribute is assigned a weight by measuring the query value distribution in the database. The rarely distribution of the query value of the attribute indicates the attribute may important for the user. The original query is then rewritten as a relaxed query by expanding each specified attribute according to its weight. The second part is result categorization. In this step, we first speculate how much the user cares about all attributes (including specified and unspecified attributes) under the query context by using the KL-divergence. Then, the categorizing attribute in each level of the tree can be determined according to its importance for the user. The most important attribute should be the categorizing attribute for the first level of the navigational tree. Lastly, the navigational tree is generated automatically and presented to the user so that the user can easily select the relevant tuples matching his/her needs. Experimental results demonstrated that the query relaxation method can achieve the Precision of 78% and 75% for UsedCarDB (used car dataset) and HouseDB (real estate dataset), respectively, and the result categorization method can also achieve the lowest total and averaged navigational costs than the existing categorization methods.

Keywords: web database, query relaxation, contextual preferences, result categorization, data distribution

1. INTRODUCTION

With the expansion of the size of Web databases and the universe use of the internet, accessing the Web database is becoming an important way for people to obtain the information. Users can only access the Web databases via their query interfaces and most of the underlying databases are usually the relational databases. In real applications, however, a user query is generally consists of several conjunctive predicates, and the traditional RDBMS (relational database management system) only support the Boolean query matching mechanism which usually causes the empty or too little answer problem when the predicates of the query are incompatible with each other or the query is very selective. In such a context, the user may hope the system can automatically relax the original query for presenting more relevant items. To deal with this problem, several ap-

Received September 5, 2015; revised September 1 & October 30, 2016; accepted November 25, 2016.
Communicated by Jiann-Liang Chen.
* This work was supported by the National Science Foundation for Young Scientists of China (61003162), the Natural Science Foundation of Liaoning Province (20170540418), and the Fund of the Education Department of Liaoning Province (LJYL018).

proaches [1-4] have been proposed and the basic idea of which is based on reducing the constraints on the original query in order to expand the scope of the query. However, most of the existing work does not consider the query value distribution in the database when relaxing the original query even though the relaxation efficiency is greatly affected by the distribution of the specified values in database.

In this paper, we propose a **Q**uery **R**elaxation and **R**esult **C**ategorization approach (hereafter referred to as QRRC) which can relax the original query and categorize the relaxed query results in a domain and user independent way. We will use the toy examples below to motivate and provide an overview of our solution.

Example 1: Consider a house selling Web database HouseDB (City, Price, SqFt, Bedrooms, Livingarea, View, Buildyear).

Table 1. An instance of HouseDB.

City	Price	SqFt	Bedrooms	Livingarea	View	Buildyear
Seattle	364900	2208	4	Burien Park	Street	1998
Seattle	226500	1160	2	Bayshore	Greenbelt	2010
Kirkland	389900	850	2	Skyway Area	Water	2006
Kirkland	556800	1450	3	Lake Forest	Water	2012
Bellevue	395000	1930	4	Richmond	Street	2008
Bellevue	466980	1608	3	Des Moines	Greenbelt	2005

Each tuple in HouseDB represents a house for sale. Based on the HouseDB the house buyer may issue the following query:

Q: HouseDB(Price between 350k and 400k $\wedge$View = *Water*)

On receiving the query, HouseDB will provide a list of a few houses with water view that are priced between $350k and $400k since there are very few houses that are priced between $350k and $400k and have the water view. In such a case, the traditional query relaxation methods will expand all basic conditions (*i.e.*, "Price between 350k and 400k" and "View = *Water*") of the query with same relaxation degree or eliminate one (or some) of them to provide relevant answer items. However, in reality there are lots of houses priced between $350k and $400k while few houses with *Water* view in the database, which indicates that the view of "water" is rarely occurred in the database. As the traditional IDF (Inverse Document Frequency) method suggested the words rarely occurred in document usually convey more information about user's needs, and thus should be weighted higher. Hence, we can speculate that the user may concern more about the attribute *View* than *Price* and thus the relaxation degree of the query criterion on the attribute *View* should be relaxed smaller than that on *Price* so that the returned answers can be more relevant to the original query.

After query relaxation, the too many answer problem (*i.e.*, "information overload") will be faced by users. To resolve this problem, two types of solutions have been proposed. The first type [5, 6] is to categorize the query results into a navigational tree, and the second type [7, 8] ranks the query results and finds the top-*k* items. The success of both approaches depends on the utilization of user preferences to filter the query results. But these approaches always assume that all users have the same preferences and they do

not consider the contexts in which the preferences appear as well. However, different users usually have different preferences on the attributes and the preferences are also associated to the specified contexts. The contextual preference, which takes the form of $\{(A_1: w_1), (A_2: w_2), \ldots, (A_m:w_m))|X\}$, meaning that attributes $(A_1, \ldots, A_m)$ are cared by the user with the weight $(w_1,\ldots, w_m)$ in the context of X.

Example 2: Consider the HouseDB mentioned above. Assume we only computed the user preferences on the attributes "SqFt" and "View" for the following two contexts:

{(SqFt: 0.26), (View: 0.09) | Price between 350k and 400k}
{(SqFt: 0.05), (View: 0.41) | Price between 350k and 400k∧City = Kirkland}

These preferences illustrate that the user preferences are associated to the specified context. In the context of a house that is priced between \$350k and \$400k, people may care more about the attribute *SqFt* than *View*, since many people wants a relative large house within this price range. In contrast, in the context of a house that is priced between \$350k and \$400k in Kirkland, people may care much more about the attribute *View* than *SqFt*. The reason is that, Kirkland is a city close to lake and most of the houses are there have the water view, thus the buyer who wants to buy a house over there may care more about view than the size (*i.e.*, *SqFt*) of the house. In this paper, we tackle the "information overload" problem for a relaxed query by proposing a categorization approach. The basic idea is to generate a navigational tree over the query results, in which the first categorizing attribute of the tree is the most important attribute for the user. Our contributions are summarized as follows:

- We propose a query relaxation method, which considers the weight of each specified attribute in the query by analyzing its specified value distribution in the database, to resolve the empty answer problem.
- We propose a novel algorithm to generate a navigational tree for categorizing the query results. This algorithm considers both the user contextual preferences and the navigational cost.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 gives the definitions of query relaxation and result categorization and presents the framework of our solution. Section 4 describes how to relax the original query while Section 5 proposes how to categorize the query results. The experiment results are presented in Section 6 and the paper is concluded in Section 7.

2. RELATED WORK

The problems of empty answers are discussed by several researches. These researches can be classified into two main categories. The first one is based on fuzzy set theory such as [2, 9, 10], in which the query criteria is relaxed by using membership functions, domain knowledge and α-cut operation of fuzzy number. The second category focuses on the development of cooperative database systems such as [1, 3, 4, 11] which

handle the query relaxation based on domain's causal structure, user feedback, taxonomy, functional dependencies, and *etc.* However, the approaches based on fuzzy sets are mainly useful in expanding the numeric query conditions while the cooperative database system usually requires the user feedback and domain knowledge. Compared with the above work, our solution is fully automatic and does not require the domain knowledge. Our solution also considers the value distribution in the dataset for relaxing the query and incorporates the user preferences for categorizing the query results.

For query results categorization, there are many researches focus on categorizing the text documents [12-14] and web search results [15, 16]. However, categorizing relational data presents unique challenges and opportunities. First, relational data contains both numerical and categorical values while the text categorization methods treat documents as bags of words. Further, our objective in this paper is to minimize the overhead for users to navigate the generated tree, which is not considered by existing text categorization methods. The work that are most similar to our categorization method are the methods proposed in [5, 6], respectively. In [5], Chakrabarti proposed a greedy algorithm to generate a category tree. This algorithm uses query history of all users in the system to infer an overall user preference as the probabilities of users are interested in each attribute. As explained in Section 1, it does not consider the diversity issue of user preferences. The different kinds of user preferences is considered by the algorithm proposed in [6], which takes advantages of the query history to infer the current user's preferences. However, the algorithm proposed in [6] does not consider the contexts in which the preferences appear. Hence, this approach may not capture the current user's preferences efficiently. Furthermore, these two approaches only consider the query history to speculate the user preferences but neglect the attribute value distribution both in the database and the query results when generating the navigational tree. Compared with the above work, our approach considers both the user's contextual preferences and the data distribution of the database and the query results.

There is also recent work on ranking query results and finding the top-k best matches from a database [17-19]. Ranking is complementary to categorization for resolving the information overload problem and we could use ranking strategy in addition to our techniques (for example, we could rank tuples in the leaf nodes of the tree).

3. PROBLEM DEFINITION AND FRAMEWORK

3.1 Problem Definitions

Consider an autonomous Web database consists of a single relational table D with m attributes $\{A_1, A_2, \ldots, A_m\}$ and n tuples $t_1, \ldots, t_n$. Let $\mathbb{R}=\bigcup_{j=1}^{m} V_j$ is a set of all attribute values, where V_j is the set of attribute values of attribute A_j.

Definition 1: (Query relaxation). Let Q be a query over D with a conjunctive predicates of the form $Q = \sigma_{\wedge i\in\{1,\ldots,k\}} q_i$, where $k \le m$, $q_i=(A_i \theta a_i)$, $\theta \in \{>, <, =, \neq, \ge, \le, \text{between}\}$, q_i is a predicate of Q. Each A_i in q_i is a specified attribute and a_i is a value (or interval) in its domain. By relaxing Q, a relaxed query which is used to find all tuples of D that show similarity to Q above a pre-defined similarity threshold $\lambda \in (0, 1]$ is obtained, *i.e.*,

$$\tilde{Q}(D) = \{t | t \in D, Sim(Q, t) \geq \lambda\}. \quad (1)$$

In this paper, our solution for relaxing a given query Q is to generate a relaxed query by reducing the constraints of Q. The relaxed query $\tilde{Q}$ should be similar to Q so that the matched tuples of $\tilde{Q}$ can satisfy the user original query intention closely. In other words, the answer tuples most related to the original query will have differences only in the least important specified attribute.

After obtaining the relaxed query results, a navigational tree will be generated for the user by considering the query context and user preferences. Let R be a set of result tuples returned by the relaxed query $\tilde{Q}$. Fig. 1 (generated by using our algorithm) shows an example of a navigational tree of the results of query $\tilde{Q}$ relaxed from the query Q presented in example 1. Then, we define a navigational tree T over R as follows.

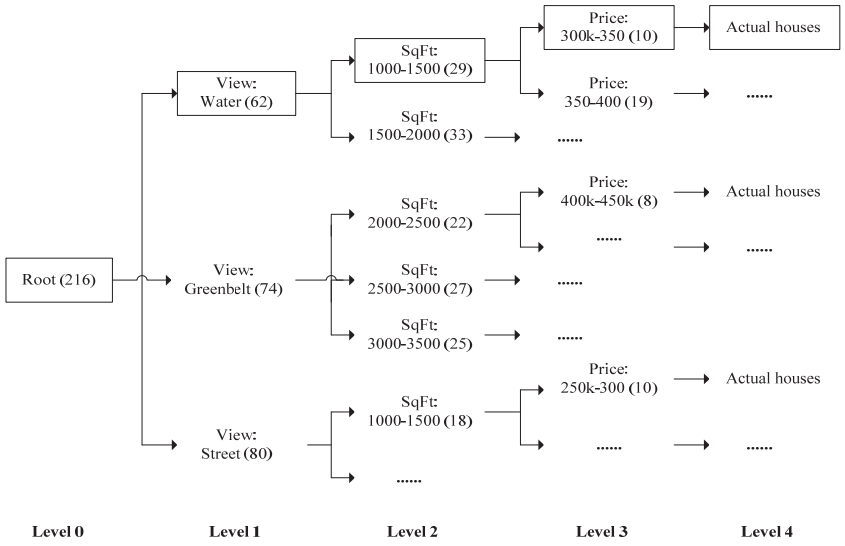

Fig. 1. An example of the navigational tree.

Definition 2: (Navigational tree) A navigational tree $T(V, E, L)$ contains a node set V, an edge set E, and a label set L. Each node $v \in V$ has a label $lab(v) \in L$ which specifies the condition on an attribute such that the following should be satisfied: (i) such conditions are point or range conditions, and the bounds in the range conditions are called partition points; (ii) v contains a set of tuples $N(v)$ that satisfy all conditions on its ancestors including itself; (iii) conditions associated with subcategory of a intermediate node v are on the same attribute (*i.e.*, *categorizing attribute*), and define a partition of the tuples in v.

Let v_j be a leaf node of T with $N(v_j)$ tuples. $Anc(v_j)$ denotes the set of ancestors of v_j including v_j itself, but excluding the root. $Sib(v_i)$ denotes the set of sibling nodes of node v_i including itself. Let K_1 and K_2 represent the unit costs of visiting a tuples in the leaf and visiting an intermediate node, respectively. Let P_j be the probability that users will be interested in the tuples of the leaf node v_j. Specifically, P_j is the product of the probability that user explores each ancestor of leaf v_j and itself, that is,

$$P_j = P(v_j) \cdot \prod_{v_i \in Anc(v_j)} P(v_i) \tag{2}$$

where, $P(v_j)$ denotes the probability that the user explores the leaf v_j and $P(v_i)$ (which will be discussed how to compute in Section 5.2) denotes the probability that user explores the intermediate node v_i that is the ancestor of v_j.

Definition 3 (Estimated navigational cost): The estimated navigational cost (*ECost*) for visiting a navigational tree is defined as,

$$ECost(T) = \sum_{v_j \in Leaf(T)} P_j(K_1 N(v_j) + K_2 \sum_{v_i \in Anc(v_j)} |sib(v_i)|). \tag{3}$$

The estimated navigational cost of a leaf node v_j consists of two terms: the cost of visiting tuples in leaf v_j (*i.e.*, $K_1N(v_j)$), and the cost of visiting intermediate nodes (*i.e.*, $K_2 \sum_{v_i \in Anc(v_j)} |sib(v_i)|$). For a given navigational tree, a user need to examine the labels of all sibling nodes to select a node on the path from the root to v_j, thus the user has to visit $\sum_{v_i \in Anc(v_j)} |sib(v_i)|$ intermediate tree nodes. The cost of visiting the root is excluded because every tree has a root. When the user reach the leaf v_j, she has to look at $N(v_j)$ tuples in v_j. For example, in Fig. 1, suppose a user is interested in the node of the top rectangle with 100% probability (*i.e.*, $P_{\text{Price:300k-350k}} = P(\text{View: Water}) \cdot P(\text{SqFt: 1000–1500}) \cdot P(\text{Price: 300k–350k}) = 1$), and then we let $K_1 = K_2 = 1$. After this, we can compute the cost of examining all the tuples under the leaf "Price: 300k-350k" is 3 (for examining the labels of the 3 first-level nodes)+2(for examining the labels of the 2 subcategories of "View: Water")+2 (for examining the labels of the 2 subcategories of "SqFt: 1000–1500")+15 (examining the tuples under the leaf "Price: 300k–350k") = 22.

3.2 Framework

The framework of our solution is shown in Fig. 2.

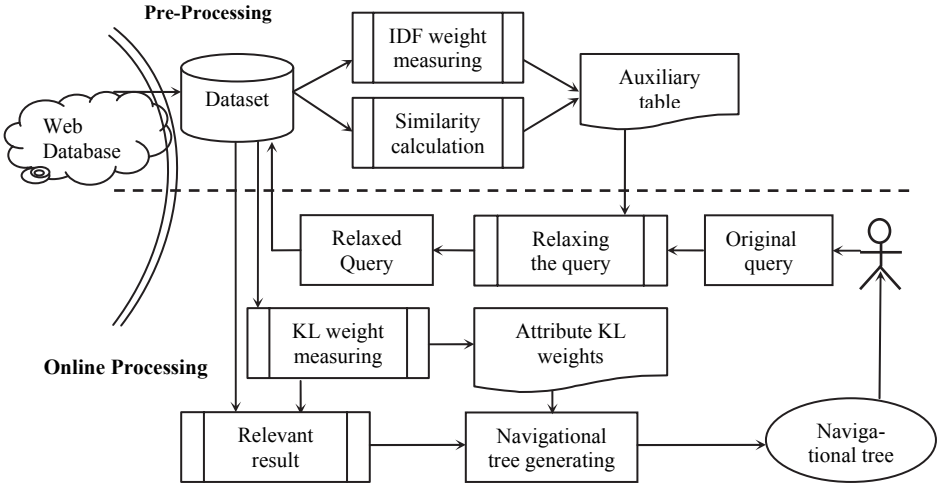

Fig. 2. The framework of query relaxation and result categorization.

The first step is to relax the original query. It computes the IDF weight for all distinct categorical values in the database and the similarity between all different pairs of categorical values during the offline time. These quantities are stored as database auxiliary tables, which are used for relaxing the original query. When a new query is coming, it first decomposes the query into several predicates, following which the IDF weight of each specified value of the query are retrieved from the IDF weight table. Then, the original query could be rewritten as a relaxed query according to the attribute weights and the attribute value similarities. Lastly, the relevant answers, which have the similarity to the original query not below the given relaxation threshold, would be returned.

The second step is to categorize the query results. Based on the relaxed query, it speculates how much the user cares about each attribute in context of the relaxed query and assigns a weight to it by using the KL (Kullback-Leibler) divergence. Then, the categorizing attribute in each level of the tree can be determined according to the weight of the attribute. The larger the attribute weight is the former level of the tree the attribute is located in. Based on the relaxed query results, it next generates a navigational tree by taking advantages of the categorizing attributes determined before and the partition criteria of the histogram construction. Lastly, this tree is presented to the user, such that the user can easily select the relevant tuples matching his/her needs by exploring the tree.

4. QUERY RELAXATION

This section describes the attribute weighting, categorical value similarity measuring and query writing methods.

4.1 Attribute Weight Assignment

4.1.1 Importance of specified categorical attribute

The traditional IDF method suggested that the words commonly occurred in the document usually convey less information about user's needs than the words rarely occurred, and thus should be weighted less. Each tuple in the database can be treated as a small document if the database only consists of text attributes. Thus, the traditional IDF method can be borrowed to solve our problem. For a point query condition "$A_i = v$", the $IDF_i(v) = \log(n/F_i(v))$ denotes the importance of attribute value v, where n is the total number of tuples and $F_i(v)$ is the number of tuples containing $A_i = v$ in the database. Consequently, the $IDF_i(v)$ can be treated as the importance of its corresponding attribute A_i.

4.1.2 Importance of specified numerical attribute

The traditional IDF method cannot be directly used for evaluating the importance of numeric values because of their binary nature. For example, given two numeric values u and v are close to each other in numeric, they would be treated as two distinct values in traditional IDF. In fact, the "IDF" of a numeric value is affected by their nearby values.

In this paper, we adopt the definition of IDF given in [20] to measure the importance of numeric attribute values. Let $\{v_1, v_2, \ldots, v_n\}$ be the values of Dom(A_i). For a

numeric value $v \in \mathrm{Dom}(A_i)$, it defined $IDF_i(v)$ as shown in Eq. (4), where h is the bandwidth parameter.

$$IDF_i(v) = \log\left(\frac{n}{\sum_{j=1}^{n} e^{-\frac{1}{2}\left(\frac{v_j - v}{h}\right)^2}}\right) \tag{4}$$

The bandwidth is $h = 1.06\sigma n^{-1/5}$, where σ is the standard deviation of $\{v_1, v_2, \ldots, v_n\}$ of the $\mathrm{Dom}(A_i)$. Actually, for v, the denominator in Eq. (4) represents the sum of "contributions" to v from every other point v_i in the $\mathrm{Dom}(A_i)$. These contributions can be modeled by Gaussian distribution, which indicate that the further v is from v_i, the smaller is the contribution from v_i to v. The importance of numeric value is also treated as the importance of its corresponding attribute.

Besides the point query, there also exists the range query which is generalized as "A_i in Ω", where Ω is a set of values for categorical attributes, or a range [lb, ub] for numeric attributes. We define the "IDF" of range queries as the minimum $\log(n/F_i(v))$ of each different value v in Ω. The importance measuring function is shown in Eq. (5).

$$IDF_i(v) = \min_{v \in \Omega} IDF_i(v) \tag{5}$$

By normalized processing, the weight w_i to attribute A_i can be calculated by Eq. (6), where k is the number of attributes specified by the query.

$$w_i = \frac{IDF_i(v)}{\sum_{j=1}^{k} IDF_j(v)} \tag{6}$$

According to the attribute weights and the threshold λ, the sub-threshold for every specified attribute can be computed by Eq. (7), where k is the number of specified attributes, λ is a given threshold, and ψ_i is the sub-threshold for specified attribute A_i.

$$\psi_{i(i=1,\ldots,k)} = \begin{cases} \frac{w_1}{\psi_1} = , \ldots , = \frac{w_k}{\psi_k} \\ \lambda = \sum_{i=1}^{k} w_i \times \psi_i \end{cases} \tag{7}$$

4.2 Attribute Value Similarity Measuring

4.2.1 Similarity of categorical attribute values

We discuss a coupling relationship measure which is an adaptation of the method proposed in [21] to capture the similarity coefficient between two categorical values. Given a pair of values binding the same categorical attribute, the intra-coupling between them indicates the involvement of their occurrence frequencies within the attribute they

belong to, while the inter-coupling means the interaction of other attributes with this attribute. The intra-coupling and inter-coupling of a pair of values are combined as their coupling relationship to reflect the relative semantic similarity.

To estimate the intra-coupling relationship between a pair of values, we use frequency of occurrence of values in the attribute domain to compute their similarity. The intra-coupling relationship between a pair of values that binding on a categorical attribute, for example A_j, of relational table D can be calculated as follows,

$$\delta_{A_j}^{IaR}(x,y)=\frac{N_{A_j}(x)\cdot N_{A_j}(y)}{N_{A_j}(x)+N_{A_j}(y)+N_{A_j}(x)\cdot N_{A_j}(y)} \tag{8}$$

where $N_{A_j}(x)$ and $N_{A_j}(y)$ denote the number of tuples that contain values x and y on the attribute A_j, respectively. The bound of the Eq. (8) is $\delta_{A_j}^{IaR}(x,y)\in[1/3, m/(m+2)]$ when $1 \le N_{A_j}(x), N_{A_j}(y) \le m$. For instance, the two values "water" and "greenbelt" in Table 1 belong to the attribute *View*, $\delta_{View}^{IaR}(Water, Greenbelt)=0.5$ since they both appeared two times in $Dom(View)$.

To estimate the inter-coupling relationship between values, we need to introduce the information conditional probability which was proposed in [23]. Given the attribute value subset $W\subseteq V_k$ of attribute A_k, and the value $x\in V_j$ of attribute A_j, the Information Conditional Probability (ICP) of W with respect to x is $P_{A_k|A_j}(W|x)$ can be computed as,

$$P_{A_k|A_j}(W\mid x)=\frac{N_{A_k}^{*}(W)\bigcap N_{A_j}(x)}{N_{A_j}(x)} \tag{9}$$

where $N_{A_j}(x)$ is defined as the same as above and $N_{A_k}^{*}(W)$ denotes the number of tuples that contain values in W on the attribute A_k. Intuitively, when given all the tuples with the value x on attribute A_j, ICP is the percentage of the common tuples whose values of attribute A_k fall in subset W and values in attribute A_j is x as well.

After this, we can define the inter-coupling relationship between two different values belong to attribute A_j based on the ICP conception,

$$\delta_{A_j|A_k}^{Is}(x,y)=\sum\nolimits_{w\in\bigcap}\{P_{A_k|A_j}(\{w\}\mid x), P_{A_k|A_j}(\{w\}\mid y)\} \tag{10}$$

where $w\in\bigcap$ denotes the intersection of the set of values on attribute A_k when the value on A_j is x and the set of values on A_k when the value on A_j is y.

According to the above discussion, the inter-coupling relationship between values x and y on attribute A_j, $\delta_{A_j}^{IeR}(x,y)$, can be computed as,

$$\delta_{A_j}^{IeR}(x,y)=\sum_{k=1,k\neq j}^{m} a_k\delta_{A_j|A_k}^{Is}(x,y) \tag{11}$$

where $\delta_{A_j|A_k}^{Is}(x,y)$ is defined in Eq. (10), $\sum_{k=1}^{m} a_k=1$ and $a_k\in[0,1]$.

Based on the intra- and inter-coupling measuring method, given a pair of values (x, y) on categorical attribute A_j, the coupling relationship between them is defined as,

$$CSim_{A_j}(x,y)=(1-a)\cdot\delta_{A_j}^{IaR}(x,y)+a\cdot\delta_{A_j}^{IeR}(x,y) \tag{12}$$

where $\alpha \in [0, 1]$ is an adjust parameter which is used to determine the weight of intra- and inter-coupling. It is clearly that the higher the coupling relationship coefficient, the more similar is the two attribute values.

4.2.2 Similarity of numerical attribute values

We propose a fuzzy set-based approach to estimate the similarity between two numerical values. In [2], the fuzzy relation "close to" is presented to estimate the similarity between two numerical values. The membership function of "close to" is showed as follows,

$$\mu_{\text{close to Y}}(u) = \frac{1}{1+\left(\frac{u-Y}{\beta}\right)^2}. \tag{13}$$

The membership function of the fuzzy number "close to Y" is shown in Fig. 3. Here larger values of β correspond to a wide curve and smaller values of β correspond to a more narrow curve. It can be seen that the membership function has the crossover points at $u = Y \pm \beta$. We will use the bandwidth h defined in Eq. (4) as the parameter β in this paper.

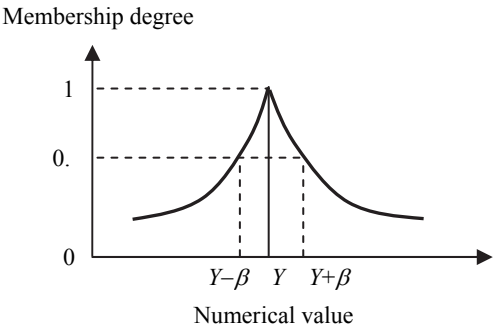

Fig. 3. The membership function of fuzzy relation "close to".

Accordingly, let $\{v_1, v_2, \ldots, v_n\}$ be the values of numerical attribute A_k and then the similarity coefficient $NSim_{A_k}(v_i, v_j)$ between v_i and v_j can be defined as,

$$NSim_{A_k}(v_i, v_j) = \frac{1}{1+\left(\frac{v_i - v_j}{h}\right)^2} \tag{14}$$

Based on the numerical value similarity, given a numerical condition $A_k = q$, let ψ_i be a sub-threshold for A_k, according to Eq. (14), we can then get the relaxation range of numeric query criterion corresponding to attribute A_k as,

$$\left[q - h\sqrt{\frac{1-\psi_k}{\psi_k}}, q + h\sqrt{\frac{1-\psi_k}{\psi_k}}\right]$$

Furthermore, the condition specified on a numerical attribute is usually an interval, *i.e.*, A_k between q_{min} and q_{max}. For such a case, the original condition can be relaxed as,

$$\left[q_{\min} - h\sqrt{\frac{1-\psi_k}{\psi_k}}, q_{\max} + h\sqrt{\frac{1-\psi_k}{\psi_k}} \right].$$

4.3 Query Rewriting

Given a query $Q = \{q_1 \wedge \ldots \wedge q_k\}$, relaxation threshold λ, attribute weights $W = \{w_1, \ldots, w_k\}$, and the attribute value similarities, the query rewriting procedure works as follows.

(i) For each predicate q_i in Q, gets the relaxed condition $\tilde{q}_i$ by extracting values that are from its corresponding attribute domain Dom(A_i) having similarity above the sub-threshold ψ_i and adding them into its query range.
(ii) By join all the relaxed predicates $\tilde{q}_i$, the relaxed query $\tilde{Q}$ is formed.
(iii) If the result of the relaxed query is not null, the algorithm terminates; otherwise, the current threshold λ decreases with the step 0.1, and the algorithm continues.

The complexity of the algorithm is $O(kn)$ where k is the number of attributes specified by the query and n is the averaged number of distinct values in the value range of all attributes.

There may too many result tuples satisfying the relaxed query. Thus, we next need to generate a navigational tree to categorize the query results.

5. QUERY RESULT CATEGORIZATION

In this section, we firstly discuss how to capture the user's contextual preferences and then present the navigational tree generating algorithm.

5.1 The Optimal Tree

The navigational tree and the navigational cost are fully defined in Section 3.1. However, in reality we want to generate an optimal tree T_{opt} such that the T_{opt} has the low information overload to the user. For the purpose, we need to do the following for each level l of the navigational tree:

(i) Determine the categorizing attribute A for the level l, and
(ii) Given the choice A of categorizing attribute for level l, for each intermediate node v in level $(l-1)$, determine how to partition the domain of values of A in v into disjoint groups and how to order those groups.

Intuitively, if the categorizing attribute in each level and the partition of domain values of the categorizing attribute can meet the user preferences closely, the cost of navigational tree will be minimized. That is, for the categorizing attributes, the categorizing attribute in the first level of the tree should be the most important attribute for the user under the given context, and so forth.

5.2 User Contextual Preferences on Categorizing Attributes

In reality the user often has different preferences under different contexts. Hence, we need to surmise the user's contextual preference when we determine the categorizing attribute for each level of the tree. The difficulty of this problem is how to determine the user's contextual preference (*i.e.*, which attributes are more important for the user under the given query context) when no user feedback is provided. To address this problem, unlike the methods proposed in [5, 6] which leverage the query history to speculate the user preferences, we assume that the user's contextual preference is reflected in the query he submitted and, hence, we use the user query as a context and hint for assigning weights to attributes (including the specified and unspecified attributes).

Since the query results binding the same values on the specified attributes, they only have differences on unspecified attributes. Thus, the problem is how two measure the correlation between the query and unspecified attributes. As pointed out by the [22], the correlation between attribute A_i and query Q can be estimated by the difference between the distributions of attribute A_i's values in the query results and their distribution in the database. The bigger the difference indicates the more A_i correlates to the query (and also the specified attribute values). KL-distance [23] is a common-used method for measuring the distribution differences. Suppose A_j is a categorical attribute of relational table D and Dom(A_j) contains values $\{a_{j1}, a_{j2}, \ldots, a_{jk}\}$, R is the result set for query Q. Then the KL-distance of A_j from D to R is:

$$D_{KL}(D \,||\, R) = \sum_{i=1}^{k} P(A_j = a_{ji} \mid D) \log \frac{P(A_j = a_{ji} \mid D)}{P(A_j = a_{ji} \mid R)} \tag{15}$$

where $P(A_j = a_{ji} \mid D)$ (resp. $P(A_j = a_{ji} \mid R)$) refers to the probability that $A_j = a_{ji}$ in D (resp. in R). If A_j is a numerical attribute, the Dom(A_j) should be first discretized into a few intervals, where each interval represents a category, and then the KL distance of A_j can be calculated by using Eq. (15).

5.2.1 Histogram construction

To calculate the KL-distance we need to obtain the distribution of attribute values over D. In this paper, we present a discretize-count method which is adapted from the algorithm proposed in [25] to build a histogram for an attribute over D. The histogram reflects the different value and its distribution of each attribute A_j in the database. Algorithm 1 shows the procedure for building a histogram for attribute A_j.

Algorithm 1: Histogram construction algorithm

Input: Attribute A_j and Dom(A_j), total number of tuples $|D|$, minimum bucket number n
Output: A histogram H_{Dj} for attribute A_j

1. **if** A_j is a categorical attribute **then**
2. **for each** distinct value a_{ji} of A_j
3. Use query condition "$A_j = a_{ji}$" to get the number of result tuples c from D
4. Add a bucket (a_{ji}, c) into H_{Dj}
5. **end for**

6. **end if**
7. **if** A_j is a numerical attribute with domain range [a_{low}, a_{up}) **then**
8. set $\lambda = |D|/n$, $low = a_{low}$, $up = a_{up}$
9. **do**
10. Use query condition "$low \le A_j < up$" to get the number of result tuples c from D
11. **if** $c \le \lambda$ **then**
12. Add a bucket (low, up, c) into H_{Dj}
13. set $low = up$, $up = a_{up}$
14. **else**
15. $up = low + (up\text{-}low)/2$
16. **end if**
17. **while** $low < a_{up}$
18. **end if**
19. **return** H_{Dj}

The algorithm builds a histogram for each attribute A_j in the preprocessing stage. The time complexity of Algorithm 1 is $O(mn)$ where m is the number of all attributes in D and n is the is the averaged distinct values in the attribute domain. A histogram H_{Rj} is also necessary to be built for A_j over the query results R to get its probability distribution over R. For each bucket of H_{Dj}, a bucket with the same bucket boundary is built in H_{Rj} and its occurred frequency is counted in R.

5.2.2 Assigning the categorizing attribute weight

After getting the histogram of A_j over D and R, the histogram is converted to a probability distribution by dividing the frequency in each bucket by the sum of bucket frequency of the histogram. That is, the probability distribution of the kth bucket of A_j for D, P_{Djk}, is

$$P_{D_jk} = \frac{c_{D_jk}}{|D|} \tag{16}$$

where c_{Djk} is the frequency of the kth bucket in H_{Dj}.

The probability distribution of the kth bucket of A_j for R, P_{Rjk}, is

$$P_{R_jk} = \frac{c_{R_jk}}{|R|} \tag{17}$$

where c_{Rjk} is the frequency of the kth bucket in H_{Rj}.

Next, for the jth attribute A_j, we assign its weight w_j as

$$w_j = \frac{D_{KL}(P_{D_j}, P_{R_j})}{\sum_{i=1}^{m} D_{KL}(P_{D_i}, P_{R_i})} \tag{18}$$

where $D_{KL}(P_{D_j}, P_{R_j}) = \sum_{i=1}^{k} P_{D_jk} \cdot \log \frac{P_{D_jk}}{P_{R_jk}}$.

Next, we use an example to show the efficiency of our attributes weights measuring method for the given contexts. Table 2 shows the use preferences on the categorizing attributes corresponding to different contexts in our experiments for the HouseDB.

Table 2. Attribute weight assignments for two different contexts.

Contexts / Attributes	*SqFt* between 2500 and 3000	City = Kirkland∧ Price between 350k and 400k
Price	0. 193	0.036
City	0.028	0.337
SqFt	0.255	0.054
Bedrooms	0.22	0.018
Bathrooms	0.105	0.015
View	0.088	0.413
Livingarea	0.065	0.068
Buildyear	0.037	0.052
Neighborhood	0.009	0.006

In Table 2, given a context "SqFt between 2500 and 3000", which means that the user prefers a large house with more bedrooms, as expected the attribute *SqFt* is assigned a large weight because it is a specified attribute and its corresponding data distribution in R and in D has a large difference. The attribute *Bedrooms* and *Price* are assigned large weights too since the large house usually priced higher and has more bedrooms.

According to the categorizing attributes weights under the given query context, the categorizing attribute in each level of the navigational tree can be determined.

5.3 Navigational Tree Generating

This section proposes the partition criteria of the attribute and the method to minimize the visiting cost. The navigational tree generating algorithm is finally presented.

5.3.1 Partition criteria

For partitioning the categorical attributes, two ways have been discussed in [5]. The one is the single-value partitioning and the other is multi-valued partitioning. In this paper we adopt the single-valued way to partition the categorical attributes. That is, assuming that there are k distinct values $\{v_1, \ldots, v_k\}$ of attribute A_j in D, we will partition D into k categories – each category corresponding to the value v_i in $\{v_1, \ldots, v_k\}$. The numerical attribute partitioning is similar to the single-value partitioning, where each bucket of the domain derived by Algorithm 1 is treated as a single-value.

5.3.2 Minimizing the visiting cost

The factor that impacts the visiting cost is the order of categories presented to the user. We next define the probability that the user exploring the navigational tree T explores category v_i. Let R' be the tuple set that satisfies all conditions on the ancestors of category v_i. Recall the histogram construction algorithm presented in Section 5.2, we build the histogram $H_{R'j}$ for attribute A_j over R' to get its probability distribution over R'.

For each bucket of H_{Dj} over D, a bucket with the same bucket boundary is also built in $H_{R'j}$. Let P_{Dji} be the probability distribution of the ith bucket of attribute A_j for D, $P_{R'ji}$ be the probability distribution of the ith bucket (corresponding to category v_i) of attribute A_j for R', k be the number of sibling nodes of node v_i including itself, and then the probability $P(v_i)$ for the category v_i can be defined as

$$P(v_i) = \frac{P_{D_j i} \cdot \log \frac{P_{D_j i}}{P_{R'_j i}}}{\sum_{l=1}^{k} P_{D_j l} \cdot \log \frac{P_{D_j l}}{P_{R'_j l}}}. \tag{19}$$

According to the Eq. (19), the exploring probability of each node v_i under the same parent will be calculated and the node with the maximum probability of the sibling nodes will be presented to the user earliest. The reason is that the larger the $P(v_i)$ is, the more the contribution of the bucket corresponding to v_i to the KL distance is, and thus the user is more interested in the values of the bucket.

5.3.3 Navigational tree generating algorithm

For generating the navigational tree, for each level l, we need to (i) determine the categorizing attribute A, and (ii) for each category v in level (l–1), partition the domain of values of A in $N(v)$ of Eq. (3) such that the information overload is low. A node v will be partitioned if v contains more than M tuples. Here, M is a given parameter and it guarantees that no leaf has more than M tuples. We now describe how a navigational tree is constructed. Since the categorizing attribute in each level of the tree is determined and the partition criteria for the values of categorizing attribute A in $N(v)$ is also proposed above, we can generate a navigational tree by using the Algorithm 2.

Algorithm 2: Navigational tree generating algorithm

Input: Database D, query results R, categorizing attributes weights, histogram H_{D_j}

Output: A navigational tree with the minimum navigational cost

```
1. Create a root node (level = 0) and add it to T
2. l = 1; // set current level to 1
3. while there exists at least one category v at level l–1 with |N(v)| > M
4.     Select A_j with the maximum weight in attributes retained and not used so far as categorizing attribute for the level l
5.     if A_j is a categorical attribute then
6.         for each value v_i of A_j in |N(v)|
7.             create a category under v and add those tuples with A_j = v_i to that category
8.         end for
9.         Compute the exploring probability of each category of v_i
10.        Sort sub-categories of v in descending order according to the exploring probabilities
11.    else // A_j is a numerical attribute
12.        for each bucket [low_j, up_j) of H_Dj for A_j in |N(v)|
13.            create a category under v and add tuples with low_i ≤ A_j < up_i to that category
14.        end for
```

15. Compute the exploring probability of each category $low_i \le A_j < up_i$
16. Sort subcategories of v in descending order according to the exploring probabilities
17. $l = l+1$; //finished creating nodes at this level, go to the next level
18. **end while**
19. **return** T

The algorithm creates the categories starting with level $l = 0$ and all next categories at level $(l–1)$ are created and added to tree T before any category at level l. A new level should be generated if there exists at least one category with more than M tuples in the current level; otherwise, the categorization is complete and the algorithm terminates. For the next level, we choose the attribute A_j which has the maximum weight in attributes retained and is not used so far as the categorizing attribute for this level. For the categorizing attribute A_j, we partition each category v at level $l–1$ with more than M tuples using the partition criteria proposed above. The subcategories of v are ranked in descending order of their exploring probabilities which can be calculated by Eq. (19). This completes the creating of nodes at level l, after which we move on to the next level.

6. EXPERIMENTS

6.1 Experimental Setup

The experiments are conducted on a computer running Windows 2007 with Intel P4 3.2-GHz CPU, and 4 GB of RAM, the RDBMS is Microsoft Sql Server 2008. We implemented all algorithms in C# and connected to the RDBMS through ADO.

Datasets: For our evaluation, we set up two datasets from two domains. The first dataset is a used car dataset **CarDB** with attributes {Make, Model, Year, Color, Engine, Price, Mileage, Transmission} containing 100,000 tuples extracted from Yahoo! Autos. The second dataset is a real estate database **HouseDB** with attributes {City, Livingarea, Price, SqFt, Bedrooms, Bathrooms, View, Neighborhood, Buildyear) containing 30,000 tuples extracted from Yahoo! Real Estate.

6.2 Categorical Attribute Value Similarity Measuring Experiment

This experiment aims at evaluating the accuracy of the categorical attribute value similarity measuring method. Table 4 shows the top-3 similar values to "Model = Camry" (in CarDB), and "View = Greenbelt" (in HouseDB) computed on different size of the datasets, respectively. Here, the parameter α in Eq. (10) is set to 0.5.

From Table 4, we found that the similarities between given attribute value and its top-3 similar values are rational and reasonable. In the CarDB, for instance, given attribute value "Camry", the values "Accord", "Corolla", and "Altima" are the top-3 similar values to it. In reality, these four model cars are quite similar to each other because they all belong to *Japan* made and are nearly identical in *Price*, *Engine*, *Color*, and other features. Additionally, even though the similarities obtained from the datasets of 5,000 and 10,000 tuples are lower than that from the dataset of 20,000 tuples with respect to a given attribute value the similarity change between them is not substantial and the relative or-

Table 4. The similarity results computed on different size of the dataset.

Categorical values	Similar values	5000 tuples	10000 tuples	20000 tuples
Model = Camry (CarDB)	Accord	0.34	0.34	0.36
	Corolla	0.29	0.30	0.32
	Altima	0.24	0.25	0.25
View = Greenbelt (HouseDB)	Greenwood	0.18	0.21	0.22
	Park	0.15	0.18	0.20
	Water	0.12	0.14	0.15

der among values is maintained, which means we can obtain the attribute value similarities on a small sample dataset during the offline pre-processing stage.

6.4 Query Relaxation Experiments

To verify the efficiency of our QR (Query Relaxation) method, we requested 5 subjects and each subject was asked to submit 3 queries for CarDB (resp. HouseDB). For each test query Q_i, a set H_i of 30 tuples, which likely to contain a good mix of relevant and irrelevant tuples to the query, is generated. We did this by mixing the top-10 result tuples that are extracted by 3 different query relaxation algorithms of QR, TOQR (a database query relaxation method by using the Bayesian causal structures discovery) proposed in [1], and AIMQ (a domain-independent approach for answering imprecise queries) proposed in [11], removing ties, and adding a few randomly selected tuples. The tuples are ranked according to their satisfaction degree to the original query. Unlike the query relaxation algorithms of IQR (an interactive query relaxation system) proposed in [3] and Taxonomy-based relaxation proposed in [4] requiring the user interaction and domain knowledge, the TOQR and AIMQ are fully automated to relax the original query and thus we choose them to obtain the relevant tuples. Finally, we presented the queries along with their corresponding H_i's to each subject in our study. Each subject's responsibility was to mark the top-10 tuples that they preferred most from the 30 unique tuples collected for each test query. We then measure how closely the 10 tuples marked as relevant by the user matched the 10 tuples returned by QR under different relaxation thresholds. The *Recall* and *Precision* metrics are used to evaluate this overlap. *Recall* is the ratio of the number of relevant tuples retrieved to the total number of relevant tuples while *Precision* is the ratio of the number of relevant tuples retrieved to the total number of retrieved tuples. In our experiments, both the relevant tuples and the retrieved tuples are 10, which makes the Precision and Recall to be equal. Fig. 4 shows the Recall&Precision of QR under different relaxation thresholds over CarDB and HouseDB, respectively.

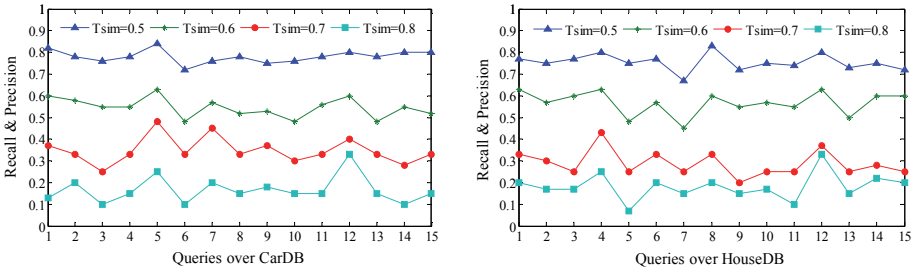

Fig. 4. Recall&Precision of QR for different thresholds over CarDB and HouseDB.

It can be seen that the Recall&Precision of QR is varied with the relaxation threshold. The lower the relaxation threshold, the higher is the recall of QR. However, this experiment mainly aims to show the averaged Recall&Precision of QR can both achieve 78% and 75% for CarDB and HouseDB, respectively. This indicates that the relaxed query generated by QR method can efficiently express the meaning of the original query. Overall speaking, the reason of QR achieved both the relatively high recall and precision is that; (i) QR speculates the weight for each specified attribute by using the IDF weighting method, which can effectively reflect the importance of the specified attribute for the user; (ii) The similarities between different categorical attribute values are reasonable since it takes both the relationship between two compared values within the attribute and the interactions come from other attributes into consideration.

6.5 Query Result Categorization Experiments

To verify the efficiency of our result categorization (referred as to RC) algorithm of QRRC, we have developed an interface that allows users to classify query results using generated navigational trees. We conducted an empirical study by asking the 10 selected subjects mentioned above to use this interface. Each subject was given the results of 10 queries over CarDB and HouseDB, respectively. We compare our RC algorithm with two other query result categorization algorithms, Cost-based algorithm proposed in [5] and C4.5 classification-based algorithm proposed in [6], respectively. For each such query, the subject was asked to go along with the trees generated by the three algorithms and to select 1-20 houses (resp. used cars) that she would like to buy.

The first metric is the total actual navigational cost defined as follows:

$$TCost(T) = \sum_{v_j \in Leaf(T)} (K_1 N(v_j) + K_2 \sum_{v_i \in Anc(v_j)} |sib(v_i)|). \tag{20}$$

Unlike the estimated navigational cost in Definition 2, this cost is the real count of intermediate nodes (including siblings) and tuples visited by a subject. For simplicity, we use equal weight for visiting intermediate nodes and visiting tuples in leaf by setting $K_1 = K_2 = 1$. Fig. 5 reports the total actual navigational cost averaged over all the subjects, for RC, C4.5 classification, and Cost-based algorithms.

Since the Pearson correlation coefficient can reflect the linear correlation between two variables, we further tested the Pearson correlation coefficient between the estimated and actual navigational cost for our algorithm (shown in Table 5).

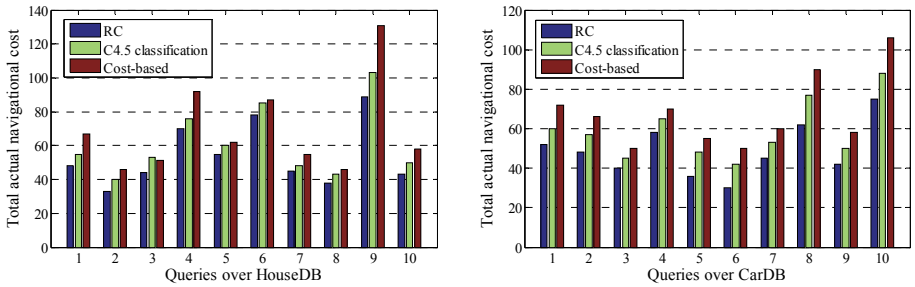

Fig. 5. Total actual navigational cost of different categorization algorithms.

Table 5. Pearson's correlation between estimated and actual navigational cost.

User	Correlation (HouseDB)	Correlation (CarDB)
Q1	0.92	0.67
Q2	0.85	0.85
Q3	0.70	0.80
Q4	0.36	–0.05
Q5	0.80	0.78
Q6	0.82	0.84
Q7	0.45	0.70
Q8	0.88	0.42
Q9	0.76	0.88
Q10	0.68	0.82
Averaged	0.72	0.67

From Table 5, it can be seen that most of the coefficients for different queries show strong positive correlation and the averaged coefficients are 0.72 and 0.67 for queries over HouseDB (resp. CarDB), which indicates there exists almost linear relationship between estimated and actual navigational costs. This demonstrated our estimated navigational cost can accurately model the information overload scenario faced by users.

The second metric is the number of relevant tuples found by a subject. In real applications, the user may find different number of relevant tuples when using different navigational trees. Generally, a good categorization algorithm should make it easy for a subject to find more relevant results. Fig. 6 reports the number of relevant tuples found by each subject using different categorization algorithms for each test query.

The third metric is the averaged navigational cost per relevant tuple found. Fig. 7 reports the averaged navigational cost of per relevant tuples found of those algorithms.

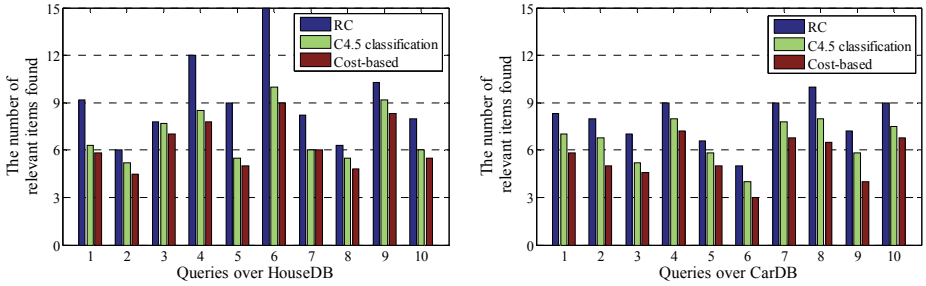

Fig. 6. The comparison of the number of relevant tuples found by per subject using different categorization algorithms for each test query.

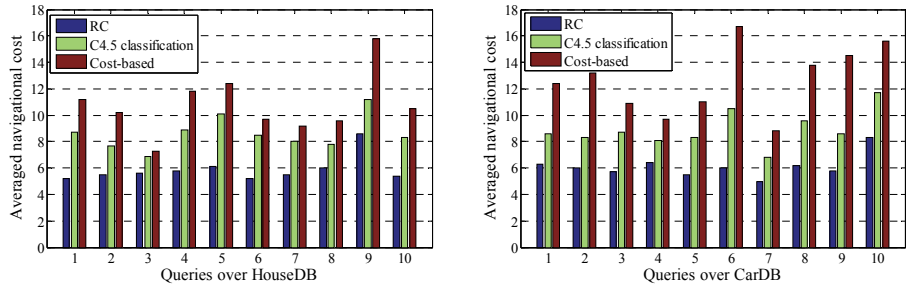

Fig. 7. Average navigational cost of different categorization algorithms for queries.

The experimental results demonstrated that the navigational trees generated by RC algorithm have both the lowest total actual navigational cost and the lowest average cost per relevant tuple found. Users have also found more relevant tuples using our algorithm than the other two algorithms. The trees generated by Cost-based algorithm have the worst results. This is expected because Cost-based algorithm ignores different user preferences. The C4.5 classification algorithm also has worse results than our algorithm. The reason is that our algorithm considers the contexts in which the preferences appear and the tree generated by our algorithm is suitable to the user's exploring style (*e.g.*, the subcategories in the same node are ordered according to their exploring probability to the current user) as well, while C4.5 classification algorithm does not. Furthermore, our algorithm uses the multi-way splits to partition the values of numerical categorizing attributes while the C4.5 classification only uses the binary split which may leads to an increase in total (resp. average) navigational cost. Besides, our algorithm considers the data distribution both in the database and the query results when constructing the navigational tree, while the Cost-based and C4.5 classification algorithms did not. Therefore, our categorization algorithm has the better performance than the existing algorithms.

7. CONCLUSIONS

In this paper, we presented QRRC, a domain independent approach for providing approximate and categorized answers for autonomous Web database queries. QRRC assigned the weights of specified attributes according to the query value distribution in the database. QRRC relaxed the original query by adding the most similar categorical values or nearby numerical values into the query criteria range. For resolving the problem of too many answers, our solution is to dynamically build a labeled and hierarchical navigational tree. By simply checking the label of a category, the user can determine whether it is relevant or not. The user only probes the interesting categories and thus the information overload can be reduced. The preliminary experimental result demonstrated that our query relaxation method can efficiently solve the empty answer problem and achieve the high Precision, the result categorization method can also have the lowest total and averaged navigational costs than the existing categorization methods.

It would be interesting to investigate (1) how to adapt to the dynamic nature and evaluate the diversity of user preferences and, (2) how does our categorization approach compare to ranking approach.

REFERENCES

1. I. Muslea and T. J. Lee, "Online query relaxation via Bayesian causal structures discovery," in *Proceedings of AAAI Conference on Artificial Intelligence*, 2005, pp. 831-836.
2. M. Hudec and M. Vucetic, "Some issues of fuzzy querying in relational databases," *Kybernetika*, Vol. 51, 2015, pp. 994-1022.
3. D. Mottin, A. Marascu, and S. B. Roy, "IQR: an interactive query relaxation system for the empty-answer problem," in *Proceedings of ACM SIGMOD International Conference on Data Management*, 2014, pp. 1095-1098.
4. D. Martinenghi and R. Torlone, "Taxonomy-based relaxation of query answering in

relational databases," *Journal of VLDB*, Vol. 23, 2014, pp. 747-769.
5. Z. Y. Chen and T. Li, "Addressing diverse user preferences in SQL-query-result navigation," in *Proceedings of ACM SIGMOD International Conference on Data Management*, 2007, pp. 641-652.
6. K. Chakrabarti, S. Chaudhuri, and S. Hwang, "Automatic categorization of query results," in *Proceedings of ACM SIGMOD International Conference on Data Management*, 2004, pp. 755-766.
7. K. Chakrabarti, V. Ganti, and J. Han, "Ranking objects based on relationships," in *Proceedings of International Conference on Extending Database Technology*, 2009, pp. 910-921.
8. A. Altman and M. Tennenholtz, "An axiomatic approach to personalized ranking systems," *Journal of the ACM*, Vol. 57, 2010, pp. 1-35.
9. R. R. Yager, "Soft querying of standard and uncertain databases," *IEEE Transactions on Fuzzy Systems*, Vol. 18, 2010, pp. 336-347.
10. P. Bhatnagar and N. Pareek, "Improving pseudo relevance feedback based query expansion using genetic fuzzy approach and semantic similarity notion," *Journal of Information Science*, Vol. 40, 2014, pp. 523-537.
11. U. Nambiar and S. Kambhampati, "Answering imprecise queries over web databases," in *Proceedings of International Conference on Data Engineering*, 2006, pp. 45-54.
12. B. Tang, H. B. He, P. M. Baggenstoss, and S. Kay, "A bayesian classification approach using class-specific features for text categorization," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 28, 2016, pp. 1602-1606.
13. F. Rousseau, E. Kiagias, and M. Vazirgiannis, "Text categorization as a graph classification problem," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics*, 2015, pp. 1702-1712.
14. W. B. Zheng, H. Tang, and Y. T. Qian, "Collaborative work with linear classifier and extreme learning machine for fast text categorization," *Journal of World Wide Web*, Vol. 18, 2015, pp. 235-252.
15. M. Laclavik, M. Ciglan, and S. Steingold, "Search query categorization at scale," in *Proceedings of ACM World Wide Web International Conference*, 2015, pp. 1281-1286.
16. M. Grbovic, N. Djuric, and V. Radosavljevic, "A large-scale semi-supervised system for categorization of web search queries," in *Proceedings of International World Wide Web Conference*, 2015, pp. 199-202.
17. A. Telang, S. Chakravarthy, and C. Li, "Personalized ranking in web databases: establishing and utilizing an appropriate workload," *Distributed and Parallel Databases*, Vol. 31, 2013, pp. 47-70.
18. A. Yu, P. K. Agarwal, and J. Yang, "Processing a large number of continuous preference top-k queries," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2012, pp. 397-408.
19. C. Y. Zhang, Y. Zhang, W. J. Zhang, and X. M. Lin, "Inverted linear quadtree: efficient top k spatial keyword search," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 28, 2016, pp. 1706-1721.
20. S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis, "Automated ranking of database query results," *ACM Transactions on Database Systems*, Vol. 28, 2003, pp. 140-174.
21. C. Wang, L. B. Cao, and M. C. Wang, "Coupled nominal similarity in unsupervised learning," in *Proceedings of International Conference on Knowledge and Infor-*

mation Management, 2011, pp. 973-978.
22. W. Su, J. Wang, and Q. Huang, "Query result ranking over e-commerce web databases," in *Proceedings of ACM Conference on Information and Knowledge Management*, 2006, pp. 575-584.
23. R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*, John Wiley & Sons, USA, 2001.

Xiaoyan Zhang (张霄雁) received her MS degree in Computer Application Technology from Northeastern University, China in 2008. She is currently an Engineer in the School of Electronic and Information Engineering at Liaoning Technical University, China. Her main research interests include spatial data mining, Web database flexible query, and city computing.

Xiangfu Meng (孟祥福) received his MS and Ph.D. degrees in Computer Application Technology from Liaoning University of Technology and Northeastern University, China in 2007 and 2010, respectively. He is currently an Associate Professor in the School of Electronic and Information Engineering at Liaoning Technical University, China. His research interests are in top-*k* query over Web database and XML data, user preference modeling, and spatial data mining.

Yanhuan Tang (唐延欢) is a master student in School of Electronic and Information Engineering at Liaoning Technical University, China. His research interests include machine learning and data analysis.

Chongchun Bi (毕崇春) is a master student in School of Electronic and Information Engineering at Liaoning Technical University, China. His research interests include spatial database keyword query and recommendation system.