

Supporting User Query Relaxation in a Recommender System

Nader Mirzadeh¹, Francesco Ricci¹, and Mukesh Bansal²

¹ Electronic Commerce and Tourism Research Laboratory, ITC-irst, Via Solteri 38,
Trento, Italy

{mirzadeh,ricci}@itc.it,

² Telethon Institute of Genetics and Medicine, Via Pietro Castellino 111, Napoli
bansal@tigem.it

Abstract. This paper presents a new technology for supporting flexible query management in recommender systems. It is aimed at guiding a user in refining her query when it fails to return any item. It allows the user to understand the culprit of the failure and to decide what is the best compromise to chose. The method uses the notion of hierarchical abstraction among a set of features, and tries to relax first the constraint on the feature with lowest abstraction, hence with the lightest revision of the original user needs. We have introduced this methodology in a travel recommender system as a query refinement tool used to pass the returned items by the query to a case-based ranking algorithm, before showing the query results to the user. We discuss the results of the empirical evaluation which shows that the method, even if incomplete, is powerful enough to assist the users most of the time.

1 Introduction

Business to consumer web sites have quickly proliferated and nowadays almost every kind of product or service can be bought on-line. In order to improve the effectiveness of user decision making support, recommender systems have been introduced in some domains. For instance, it is very popular the books recommender system included in amazon.com or the movie recommender system MovieLens [1]. In a previous paper we have introduced Trip@dvice, a travel planning recommendation methodology that integrates interactive query management and case-based reasoning (CBR) [2]. In this methodology, product research is supported by an interactive query management subsystems that cooperates with the user and provides information about the cause of query failure and its possible remedies. The goal is to help the user to autonomously decide what is the best compromise when not all his wants and needs can be satisfied. CBR supports the modelling of the human/computer interaction as a case and is exploited to extract, from previously recorded recommendation sessions, useful information that enable to intelligently sort the results of a user's query.

In this paper we focus on the Trip@dvice relaxation algorithm and its evaluation in practical usage. Interactive query management, and in particular dealing

with failing queries has been addressed in a number of researches in the area of cooperative database [3–5]. Chu et al. suggested to explore an abstraction hierarchy among values of a feature/attribute, and to use that knowledge for moving along the hierarchy for relaxation or tightening the result set [3]. Godfrey extensively studied the cause of failure of boolean queries, and proposed an algorithm for query relaxation [5].

Our new query relaxation algorithm extends that presented in [6], introducing the notion of abstraction-hierarchy. This is a relationship among features describing the product and not, as in [3], a partition of a feature domain (grouping of values). Moreover our approach is not limited to boolean constraints, as that discussed in [5], as we address the problem of relaxing range constraints on numeric attributes. In principle our goal is to relax the minimum number of constraints for a given query, such that a non void result is returned. But, finding such relaxed query is in general an NP-hard problem [5], hence we focussed on a simpler, incomplete but practically feasible and effective approach.

In this paper we first present the motivation for using interactive query management (IQM) in recommender systems and then we describe the architecture of the IQM module implemented in Trip@dvice. Then we present the empirical evaluation of IQM, that was conducted by exploiting log data derived from the evaluation a recommender system based on Trip@dvice [2]. The results show the method is powerful enough to find at lease one successful relaxed query most of the time.

2 Interactive Query Management

In previous researches on knowledge-based recommender systems the problem of dealing with the failure of an over-constrained query was typically addressed by exploiting similarity-based retrieval [7, 8]. In Trip@dvice [2], as well as in other proposals [9], it is argued that the system should not autonomously determine the best attainable approximate match, as in a similarity-based retrieval, but should actively support the user and let him understand what is the best relaxation/compromise. Hence in Trip@dvice when a query fails to return any item a dedicated interactive query management component (IQM in Figure 1) suggests some refinement(s) in terms of query constraints to discard or change.

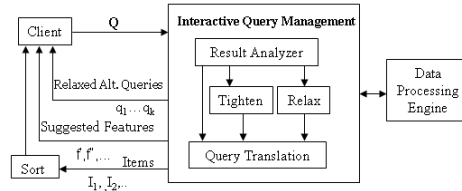


Fig. 1. Intelligent Mediator architecture.

A user will typically interact with IQM through a Graphical User Interface, submitting a query and getting as reply a reasonable set of items. If this cannot be achieved then IQM suggests to the user some refinements to the failing query. Two types of failures are considered: the query returns either too many results or no result at all. When a query returns an empty set, IQM tries to find some relaxed alternative queries that change or remove the minimum number of constraints in the failing query and would make the subquery to produce some results. IQM reports these alternative options to the user so that he can decide what is the best relaxation (from his point of view) and opt for this with the certainty that the query will now return some result. Conversely, in case too many items would be returned, a set of candidate features are suggested as those that should be constrained in addition to those already included in the original query (this last topic is not discussed here, see [6]).

The query language supported in Trip@dvce is simple but expressive enough to cope with standard web forms. Let $X = \prod_{i=1}^n X_i$ be an item space (e.g., a set of hotels). We shall refer to a feature of X , as f_i with its associated domain X_i for all i , $1 \leq i \leq n$. A query, say q is obtained by the conjunction of atomic constraints; $q = c_1 \wedge \dots \wedge c_m$ (or simply $q = \{c_1, \dots, c_m\}$), where $m \leq n$ and each c_k ($1 \leq k \leq m$) constrains the feature f_{i_k} . Each constraint is defined as follows

$$c_k = \begin{cases} f_{i_k} = \text{true} & \text{if } f_{i_k} \text{ is boolean} \\ f_{i_k} = v & \text{if } f_{i_k} \text{ is nominal} \\ f_{i_k} \in [l, u] & \text{if } f_{i_k} \text{ is numeric} \end{cases} \quad (1)$$

where $v, l, u \in X_{i_k}$.

Note that the query language we consider here does not support inquiries across catalogues, because we assume that a catalogue is a custom view that is possibly built over a set of joint tables.

We now focus on the Relax module in Figure 1. We shall refer to a relaxed version of a query by calling it simply a *subquery*, and a *successful subquery* if it returns a non-empty result set.

Example 1. Let $q : \{2 \leq \text{category} \leq 3, [\text{parking} = \text{true}], [30 \leq \text{price} \leq 50]\}$ be a query, then $q' : \{[\text{category} \geq 3], [\text{price} \leq 35]\}$ is a subquery of q , where the constraint on parking is relaxed.

The relaxation method makes use of the notion of *feature abstraction hierarchy*, i.e., a hierarchy among some features of an item space. For example, assume the *accommodation* item space contains the *category* (i.e., 1-star, 2-star, ...), *price* and *parking* features. Then we say that the feature *category* is more abstract than the *price* because the knowledge of the price greatly reduces the uncertainty over the category (i.e the conditional entropy of the category given the price is low [10]). The idea is to use such relationships in the relaxation process to sort a set of related constraints and start relaxation on the constrained-feature with lowest abstraction level. If this relaxation does not produce any result, then this constraint is removed and the constraint on the feature with next-lower abstraction is tried.

More formally, we denote with $FAH = \{F^1, \dots, F^k\}$ a collection of feature abstraction hierarchies, where each $F^i = (f_1^i, \dots, f_{n_i}^i)$ is an abstraction hierarchy,

i.e., an ordered list of features in X (comparable features). We mean that f_j^i is more abstract than f_j^i iff $j < l$, and there are no common features in two different F^i s, i.e. features in different F^i are not comparable.

Figure 2 depicts the algorithm used in Trip@dvice to find relaxing sub-queries of a failing query.³ The constraints in the query $q = \{c_1, \dots, c_m\}$ are partitioned in the set $CL = \{cl_1, \dots, cl_{m'}\}$, where each cl_i is a ordered list of constraints of q , such that the features in cl_i belongs only to one F^j (line 1). If the constrained feature in any constraint c does not belong to any abstraction hierarchy, then a singleton list, containing only c , is built. So for instance, let $q = \{c_1, c_2, c_3\}$ be the query as in Example 1, and let the feature abstraction hierarchy be $FAH = \{(category, price)\}$, then $CL = \{(c_1, c_3), (c_2)\}$ is the partition of q according to this FAH . The proposed algorithm outputs at most two subqueries, one for each element in CL . The first subquery is obtained by relaxing a constraint in (c_1, c_3) the second by relaxing c_2 . To find the first subquery, the algorithm first checks if the relaxation of c_3 (price) produces a successful subquery, otherwise it removes c_3 and relaxes also c_1 .

More precisely, the loop at line 3 tries to relax the set of related constraints (i.e., $cl \in CL$) while keeping the rest of the constraints in q unchanged. The related constraints in cl are relaxed starting from the one on the feature with lowest abstraction by the *for* loop at line 6. ch is a list containing all the modified versions of the current constraint, and it is initialized with c at line 8. The variable *relax* is a flag that indicates whether a wider or shorter range should be tried for a numerical constraints. It is initially set to *true*.

```

q = {c1, ..., cm}: a query to be relaxed.
RelaxQuery(q)
1 CL ← Partition constraints in q according
  to the defined FAH;
2 QL ← ∅; % the list of subqueries
3 for each cl ∈ CL do
4   q' ← q;
5   suggest ← ∅; % a subquery suggestion
6   for j = |cl| to 1 do
7     c ← get jth constraint from cl
8     ch ← {c} % constraint history
9     relax ← true
10    do
11      q' ← q' - {c}
12      c ← ModifyRange(c, ch, relax)
13      while ( Analyse(q', c, ch, relax, suggest) )
14      if suggest ≠ ∅ then
15        QL ← QL ∪ suggest;
16        j ← 0; % no need to go to higher abstr.
17      end:if
18    end:for
19 end:for
20 return QL

ModifyRange(c, relax, ch)
1 c' ← c;
2 if c is numeric then
3   if relax then
4     c' ← IncreaseRange(c);
5   else
6     c' ← DecreaseRange(c, ch);
7   end:if
8 return c'

Analyse(q', c, ch, relax, suggest)
1 retValue ← false;
2 if (c ≠ null and
  length(ch) < max.length) then
3   if c is numeric then
4     retValue ← true;
5     q' ← q' ∪ {c};
6   end:if
7   n ← Count(q');
8   if (n = 0 and relax = false) then
9     retValue ← false;
10  else if (n = 0) then
11    ch ← ch ∪ {c};
12    relax ← true;
13  else if n > α then
14    relax ← false;
15  if n > 0 then
16    suggest ← {(q', n)};
17 end:if
18 return retValue

```

Fig. 2. Relaxation algorithms.

³ Variables are in *italic* and literals are in sans serif

Then the *do-while* loop at lines 10-13 tries different ranges for a numerical constraint, or it iterates only once for a non-numerical constraint. The `ModifyRange` function modifies only the numerical constraints. It returns a modified version of the input (numeric) constraint depending on the *relax* flag. The returned constraint contains either a wider or shorter range than the one specified in c depending on the value of *relax*. It returns `null` if increasing (decreasing) a range would not increase (decrease) the result size of the subquery. For instance, if the current range specifies the whole range of values in the catalogue for the constrained feature, then increasing the range would not improve the result size of current subquery.

Let $c : [l \leq f_i \leq u]$ be current range constraint. The `IncreaseRange` function relaxes c to $[l - \delta] \leq f_i \leq [u + \delta]$, where

$$\delta = \begin{cases} 0.1(u - l) & \text{if } u > l \\ 0.1(v_{max} - v_{min}) & \text{if } u = l \end{cases} \quad (2)$$

and v_{min}, v_{max} are the minimum and maximum values of feature f_i in the catalogue, respectively.

Example 2. (Example 1 cont.) Assume q returns an empty set. If we call `RelaxQuery` to find all the relaxed subqueries of q , the `IncreaseRange` will be called in turn to find a new wider range for *price*, and it would return $[28 \leq price \leq 52]$.

The `DecreaseRange` will find a range between that range that caused an empty result set and the current one that makes the subquery to return too many results. Let $[l_1, u_1]$, and $[l_2, u_2]$ be two such ranges. Using this method to shorten $[l_2, u_2]$, it would return a new constraint specifying the range $[l', u']$ where $l' = (l_2 - l_1)/2$ and $u' = (u_2 - u_1)/2$.

Example 3. (Example 2 cont.) Assume trying the wider range $[28, 52]$ for the *price* returns too many items by the corresponding subquery q' (i.e., $\text{Count}(q') > \alpha$). The next call to the `ModifyRange` will call the `DecreaseRange` in order to reduce the current range, and it will return the constraint $[29 \leq price \leq 51]$.

The `Analyse` procedure determines whether or not the maximum number of attempts to find a suitable constraint for a numeric feature has been reached. If not, it examines the result size (n) of the current subquery. If the result size is 0, it sets the *relax* flag to `true` to indicate a wider range should be tried, otherwise if the current range has made the subquery to produce a large result set, then the *relax* is set to `false` to indicate a shorter range has to be tried. If at any stage the subquery produces some results, then the method creates a new suggestion (i.e., *suggest*) which is a set containing a pair made of the subquery and its result size (line 16, in the `Analyse`).

After the *do-while* loop (lines 10-13), if there is any suggestion, then there is no need to relax the constraint on feature with higher abstraction, and hence the loop at line 6 terminates and the suggestion is added to the return set (QL).

The algorithm terminates when all the lists $cl \in CL$ of q are tried. Obviously, the running time of `RelaxQuery` is $O(|q|)$ where $|q|$ is the number of constraints in the query q , since each constraint is considered a constant number of times.

It is worth to note that the RelaxQuery returns at most the same number of successful subqueries as there are elements in the partition set of the query q , where each subquery represents a compromise (i.e., relaxation of a subset of constraints in the query) the user has to make in order to receive some results. Hence, it is up to the user to judge which subquery is the best one according to her preferences and constraints. For instance, if the RelaxQuery returns two subqueries, one that relaxes the constraints on price and the category, and another that relaxes the parking feature, then the user can choose either view more expensive hotels or to miss the parking.

3 Evaluation

We developed a prototype, based on Trip@dvice [2], and empirically evaluated that with real users. In fact two variants of the same system (*NutKing*⁴) were built. One supports intelligent query management (IQM) and intelligent ranking based on cases (*NutKing+*), and the other without those functions (*NutKing-*). In fact *NutKing+* used a previous version of the relaxation algorithm, that did not use the notion of feature-abstraction hierarchy [6]. The subjects had randomly been assigned to one of the two variants.

Table 1 shows some objective measures relative to the IQM. It includes the average values and standard deviations of the measures taken for each recommendation session. Values marked with * (**) means a significant difference at the 0.1 (0.05) probability level, according to an unpaired t-test. Relaxation was

Table 1. Comparison of NutKing+ and NutKing-.

Objective Measure	NutKing-	NutKing+
queries issued by the user	20.1±19.2	13.4±9.3*
number of features in a query	4.7±1.2	4.4±1.1
results size per query	42.0±61.2	9.8±14.3**
system suggested query relaxations	n.a.	6.3±3.6
# of times the user accepted query relaxations	n.a.	2.8±2.1

suggested by the system 6.3 times per session, and this means that approximately 50% of the queries had a “no result” failure. The user accepted one of the proposed subqueries 2.8 times, i.e. almost 45% of the time. We consider this a good result, taking into account the user behavior that is often erratic and not always focussed in solving the task.

We now provide a more detailed description of the queries issued to each of the five catalogues used in the experiment and of the performance of the RelaxQuery algorithm. In particular we shall show how often RelaxQuery can effectively assist users.

Five catalogues/collections are considered in NutKing: accommodations, cultural attractions, events, locations, and sport activities. We do not consider here the *cultural attractions* catalog since it has only three features and relaxation of 1-constraint in a failing culture-query has always produced some results.

⁴ <http://itr.itc.it>

We mined the log file of users’ interactions in the empirical evaluation, and extracted all the users’ queries submitted to different catalogues in NutKing±. We grouped queries according to their type (i.e. the catalogue they queried) and the number of constraints they contained. Let Q^j denote the set of queries submitted to a catalogue where each query contains j constraints. The log also contained the result size of each query, which enabled us to determine the set of failing queries, i.e., those that had a void result set. Let $FQ^j \subseteq Q^j$ denote the set of such failing queries. Then we ran (again) the RelaxQuery algorithm on each query $q \in FQ^j$ to compare the subset of failing queries, say SQ^j that the RelaxQuery could find a successful subquery, with and without the notion of feature-abstraction-hierarchy (FAH) (Table 2). The relaxation algorithm that does not exploit FAH is described in [6]. In total using *FAH* the proposed algorithm was able to find a successful subquery 83% of the cases (232/279), whereas the same algorithm not using *FAH* was successful 61% (172/279). It is worth noting that, this is obtained by loosing a property of the previous method, i.e., the capability to find all the successful subqueries that relax 1-single constraint. In other words there are cases in which using FAH two constraints in a hierarchy are relaxed even when only one is necessary. This happen when relaxing the more abstract and still keeping the lees abstract does give some results. But, in practice, using FAH we can find a successful subquery in more cases.

Table 2. Queries submitted to the catalogues.

Catalogue	$\sum Q^j$	$\sum FQ^j$	$\sum SQ^j$	$\sum SQ^j$ FAH
Accommodation	186	112	73	94
Location	116	64	29	50
Event	92	57	39	52
Sport	102	49	31	46
Total	496	279	172	232

The figures 3(a)-(d) compare the size of the sets Q^j , FQ^j , and SQ^j for the location, accommodation, event, and sport catalogues, respectively, when *FAH* is used.

This shows that RelaxQuery can always find a successful subquery when the user query contains up to three constraints. This number is even higher for the queries submitted to event and lodging catalogues. In particular, considering the accommodation catalogue, we observe that more than half of queries failed to return any item, and the RelaxQuery could find a subquery 83% of the time (Table 2). We also see the method performs worse for the location-queries. One reason for this is location has more features than the others, and as the number of constraints in a query increases, the likelihood increases that a failing query cannot return any result even by *removing one* constraint. In other words, the more constraints a query has, the more likely it is to have a situation that 3 constraints are inconsistent such that no item can satisfy those constraints and we have to remove at least 2 constraints in order to receive some results. This is the case for many queries of the location catalogue with 8 constraints or more.

Finally, we want to illustrate how difficult it would be for a user to refine autonomously her failing query. We note that we did not mined the log data coming

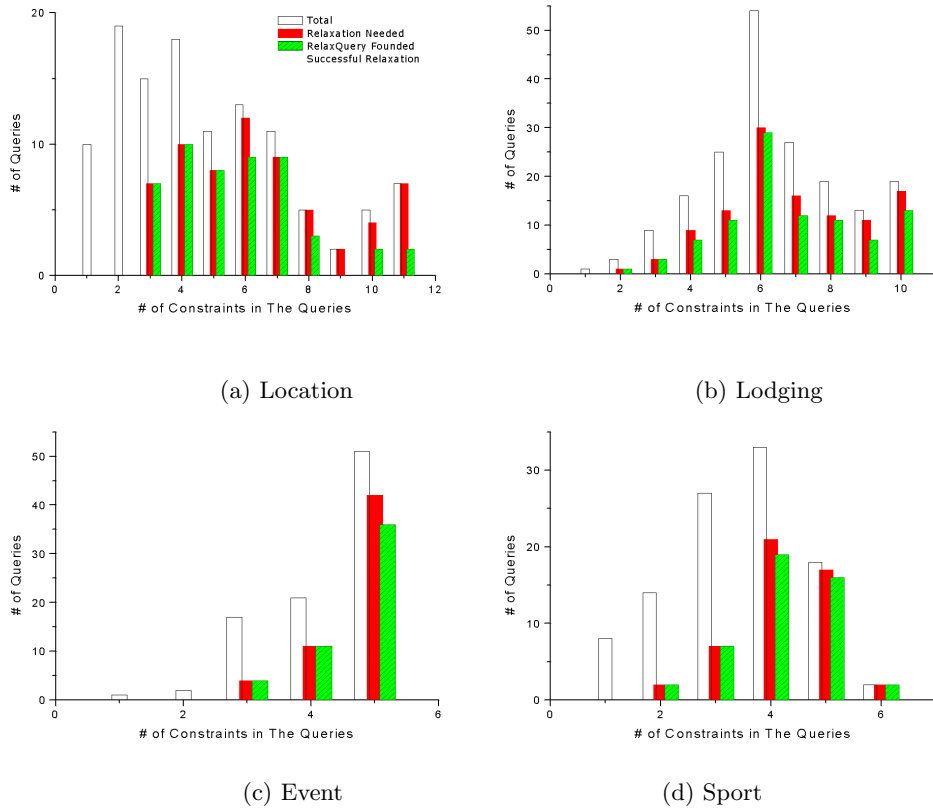


Fig. 3. Behavior of the RelaxQuery algorithm for different catalogues.

from NutKing-, to measure if and how the user relaxed the failing queries (without system assistance). The discussion here is hypothetical, and is aimed at understanding, in the worse case, the number of query change attempts (constraint removals, that are needed to receive some results discarding the minimum number of constraints. Let $q = \{c_1, \dots, c_m\}$ be a query, and assume the RelaxQuery method can find k successful subqueries. The *maximum* number of attempts the user needs in order to find a successful change to her failing that relaxes only one constraint is $m - k + 1$. In fact, if a user query receives zero results, then in a second attempt, the user must remove one constraint, and submit the new query. If this again fails he must put back the *removed* constraint and discard another constraint, and submit the new query. He must proceed until receives some results, which could be, in the worse case at the $(m + k - 1)^{\text{th}}$ attempt. In fact, this number will be even larger if q contains some range constraint, because different ranges should be tried. In practice this is not feasible for a user.

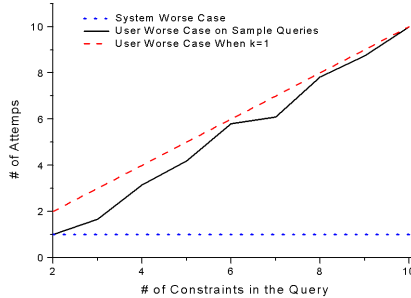


Fig. 4. Steps needed to find a relaxation.

4 Related Work

The research on failing queries goes back to the work of Kaplan [11], where he described a system that supported a cooperative query answering to queries expressed in natural language. He argued that it is more informative to let the user understand the cause of failure, and provide answers that partially satisfy the query, than just reporting a void answer like the empty result set.

Value abstraction was introduced in the CoBase system [3], where an abstraction hierarchy among feature values is built, and the relaxation operators use it to move up and down the hierarchy. In our approach, a hierarchy is a relationship between features (e.g., country \succ county \succ city), derived from domain knowledge, and is exploited to find successful sub-queries of a failing query. While building a values abstraction hierarchy is quite complex and costly in CoBase, in our approach the feature hierarchy can be achieved with minimal effort.

Gaasterland et al. described in [4] a logic-based approach to develop a cooperative answering system that accepts natural language queries, and can explain the cause of any failing query. Godfrey has extensively investigated the problem of identifying the cause of a failing boolean-query [5]. He has derived an algorithm (called ISHMAEL) to find successful maximal subquery (called XSS), i.e., not contained in any other successful subquery. He shows that finding only one XSS can be done in linear time proportional to the number of constraints in q , but finding all XSS is intractable. The major difference between RelaxQuery and ISHMAEL relates to the search mode. While the latter does a depth-first-search to find an XSS subquery, the former takes a breadth-first-search to find a subquery that relaxes minimum number of constraints, hence must be incomplete to avoid the combinatorial explosion of search states.

McSherry [9] has approached the relaxation problem differently by looking at the products that do not satisfy the query. For each product in the catalogue the set of attributes (called compromise set) that do not satisfy the query is computed, and a case representative (i.e., with highest similarity to the query) is put in a set called *retrieval set*. The problem with this approach is the retrieval set may contain up to $2^{|q|}$ elements.

5 Conclusions and Future Work

In this paper we presented a linear time incomplete algorithm for query relaxation. If it finds a solution this is maximal with respect to the number of constraints kept in the successful subquery found. Thus, since a query represents user's preferences, the algorithm relaxes as few as possible user's preferences. This algorithm is simple enough to be easily integrated into any eCommerce application. An empirical evaluation of the algorithm was presented, and showed the algorithm is powerful enough to recover from a failure situation most of the time.

In the future we plan to introduce *user modelling* principles that would enable to assign weights to user's preferences such that the preferences with lowest weight will be relaxed first. Moreover, we are extending RelaxQuery to cope with the general case, i.e., when more than one constraint must be relaxed to find a successful subquery.

References

1. Schafer, J.B., Konstan, J.A., Riedl, J.: E-commerce recommendation applications. *Data Mining and Knowledge Discovery* **5** (2001) 115–153
2. Ricci, F., Venturini, A., Cavada, D., Mirzadeh, N., Blaas, D., Nones, M.: Product recommendation with interactive query management and twofold similarity. In Aamodt, A., Bridge, D., Ashley, K., eds.: ICCBR 2003, the 5th International Conference on Case-Based Reasoning, Trondheim, Norway (2003) 479–493
3. Chu, W.W., Yang, H., Chiang, K., Minock, M., Chow, G., Larson, C.: Cobase: A scalable and extensible cooperative information system. *Journal of Intelligence Information Systems* **6** (1996)
4. Gaasterland, T., Godfrey, P., Minker, J.: Relaxation as a platform for cooperative answering. *Journal of Intelligent Information Systems* **1** (1992) 293–321
5. Godfrey, P.: Minimization in cooperative response to failing database queries. *International Journal of Cooperative Information Systems* **6** (1997) 95–159
6. Ricci, F., Mirzadeh, N., Venturini, A.: Intelligent query management in a mediator architecture. In: 2002 First International IEEE Symposium "Intelligent Systems", Varna, Bulgaria (2002) 221–226
7. Burke, R.: Knowledge-based recommender systems. In Daily, J.E., Kent, A., Lancour, H., eds.: *Encyclopedia of Library and Information Science*. Volume 69. Marcel Dekker (2000)
8. Kohlmaier, A., Schmitt, S., Bergmann, R.: A similarity-based approach to attribute selection in user-adaptive sales dialogs. In: *Proceedings of the 4th International Conference on Case-Based Reasoning*. Volume 2080 of LNAI., Vancouver, Canada, Springer (2001) 306–320
9. McSherry, D.: Similarity and compromise. In Aamodt, A., Bridge, D., Ashley, K., eds.: ICCBR 2003, the 5th International Conference on Case-Based Reasoning, Trondheim, Norway (2003) 291–305
10. MacKay, D.J.C.: *Information Theory, Inference and Learning Algorithms*. First edn. Cambridge University Press (2003)
11. Kaplan, S.J.: Indirect responses to loaded questions. In: *Proceedings of the theoretical issues in natural language processing-2*. (1978) 202–209