Intelligent Query Management in a Mediator Architecture

Francesco Ricci, Nader Mirzadeh, Adriano Venturini

Abstract— **This paper presents a new technology aimed at supporting the user in the process of building a query to a catalogue of products. It is shown how a mediator architecture can be extended to support the relaxation or tightening of query constraints when no or too many results are retrieved from the catalogue. This paper presents the architecture and the basic algorithms of this Intelligent Mediator. An application of this technology to a recommendation system for travel products selection is also illustrated.**

Index Terms— **Mediator architecture, query refinement, decision support systems, recommender systems.**

I. Introduction

All major eCommerce web sites allow the user to search in catalogues of products and get recommendations. These web sites ask the user constraints or preferences related to the products, and then searches for those items that match the user's request.

But there are some inherent limitations to this approach. First of all, to maintain simple and not tedious the man/machine interaction, the web form includes only a limited number of product features to be constrained by the user selection. But at the same time the form must try to catch all the user relevant preferences, which may depend on the user's characteristics or on the very nature of the searched product. For instance, in a commercial hotel reservation system, hotel and room descriptions may easily count hundred of features. This kind of problems are normally managed by allowing the user to choose between at least two forms, one typically quite simple, and another one that shows the complete list of product features.

Second, when the results of a user request are searched in the catalogue two limit situations could happen: there could be too many results or, on the contrary, no result. These situations occur quite often, even if sometimes the user does not realize it. In fact, for instance, when too many results match the user constraints a common used "trick" is to present only a few of them, by automatically filtering some of the results. This filtering technique is typically not personalized, and this will tamper with the goodness of the recommendation.

In our view the available solutions to these problems are not completely satisfactory. The tradeoff between maintaining the user interface simple and at the same time catching all the user needs is resolved in favor of the simplicity. This is correct but it leaves the space open to more sophisticated solutions.

F. Ricci, N. Mirzadeh and A. Venturini are with the eCommerce and Tourism Research Laboratory, ITC-irst, via Sommarive 18, 38050 Povo, Italy. E-mail: {ricci,mirzadeh,venturi}@itc.it

In this paper we present a solution that relies on the observation of how these problems are managed in a "real" shop with a "real" human to human interaction (customer/clerk). The major difference in this case relies on the fact that human to human interactions is typically interactive, and the initial customer enquire might be refined in many stages with the help of the clerk. For instance, in a travel agency a hotel selection search could proceeds as follows.

- **Customer:** *I'd like to find an hotel with room-price < 100, with parking and Internet connection*
- **Clerk:** *I'm sorry there is no match. BUT: there are several hotels with room-price < 150, with parking and Internet connection OR one hotel with room-price = 96, with Internet connection but with no parking*
- **Customer:** *let me see this cheap option* [he sees it]
- **Customer :** *now let me see the other more expensive solutions*
- **Clerk:** *would you like to have a restaurant in the hotel?*
- **Customer:** *yes* [he gets 10 suggestions]

What emerges from this description is that the clerk not only passively replies to the user, but also suggests modifications to the user's query, driving actively the dialogue.

The simulation of this behaviour has been the goal of many "intelligent" systems, which are based on Artificial Intelligent, initially in the form of expert systems and now as recommender systems [1], [2], [3]. Along this line of research we have built a recommender system for supporting travel planning [4], whose central component is an Intelligent Mediator, that extends a classical query manager and mediator [5], [6] with a set of "intelligent" functions that will help the user in the process of selecting a product from a catalogue.

This paper focusses on the Intelligent Mediator component, and in particular on its logical architecture and the basic algorithms used for supporting query constraint relaxation and tightening. A description of the more classical data integration and mediator function are described in [7]. The Intelligent Mediator helps the user to reformulate the query exploiting both summary knowledge about the distribution of the data and a simple "decision model" that determines product's features relevance during the selection task.

The paper is organized as follows. Section II presents the main logical architecture of the Intelligent Mediator, the mathematical model of the items contained in the catalogues and the query language. Section III illustrates the relaxation process managed by the mediator and algorithms exploited when a query returns no items. Section IV illustrates how the Mediator deals with the opposite situation, i.e., too many items are retrieved and it is necessary to further tighten the query constraints. In the final Section we resume the results presented and point out an application that exploits this component as part of a travel planning recommender system.

II. Interactive Query Management

In this Section we describe the basic data structures and the functions designed to search in a catalogue of items. We first describe the item structure and then the basic architecture for item search.

The figure 1 shows the main logical architecture and the components designed to help the user to satisfy his information goal. The Client application (typically a Graphical User Interface) interacts with the Intelligent Mediator (IM) that, processing the input query, produces some reasonable set of items that satisfy the query constraints, or when this is not achievable, suggests some refinements to the query. This intelligent mediator in turn interacts with a data processing engine that provides physical data storage, processing, and some utility functions needed by IM.

The user's information needs are encoded as a query, say Q, and the reply of IM to Q could take the form of a set of items or a new set of queries. It is a set of items when the Intelligent Mediator decides there is a satisfactory result for the query in the data, otherwise, in case a "failure" situation occurs, it suggests a new set of alternative queries that, tightening or relaxing some of the query constraints, might produce a satisfactory results. More precisely, when a query Q is sent to IM, this is translated, according to some mapping rules, into a format that is processable by the external Data Processing Engine. The result, computed by the Data Processing Engine, is fed back to IM that analyzes it. Three situations are possible: no result, a convenient number or too many results are found. In the second case the results are sent to the client. In the other cases the Relax and Tighten modules are used to determine a new set of queries. This last step can require a further interaction with the Data Processing Engine to ensure that the new set of queries do solves the failure of the initial query. In the following sections this process is further detailed.

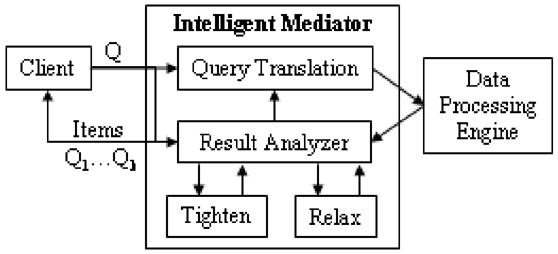

Fig. 1. Intelligent Mediator architecture.

A. Item Representation

We shall denote with X an item space (of a particular type). An item space X models a single type of items and it is a vector space $X = \prod_{i=1}^{n} X_i$, i.e., an item of type X is represented as a $n-$dimensional vector of features (we use the word "feature" as a synonym of an item attribute or item attribute space of values). An item space, in a eCommerce application, represents a catalogue of product/services accessed through the web. For instance, in our application to travel products recommendation, the hotel "Majestic, located in Pza. Navona, in Rome, whose single room costs 125 Euro per night, has category 3, has lift and Air Conditioning" can be described as the vector $x = (Rome, PzaNavona, Majestic, 3, 125, \top, \top) \in Location \times Address \times Name \times Category \times Cost \times AC \times Lift$, where, e.g "Rome" is the feature value of the "Location" feature. We consider three general types of features:

- *Finite Integer:* A finite integer feature contains a finite set of elements among which a relation of total ordering is defined (denoted with $<$). Examples of finite integer features are : $Level = \{beginner, intermediate, advanced\}$, $Category = \{0, 1, 2, 3\}$. The order relation states that, $0 < 1 < 2 < 3$, or $beginner < intermediate < advanced$.
- *Real:* A real feature takes values in an interval of the real numbers (e.g. $Cost$). Real features have a total order relation as well.
- *Symbolic:* A symbolic feature contains a finite number of elements. Examples of symbolic spaces are: $AC = \{\top, \bot\}$ the boolean space; or $Location = \{Rome, Milan, Venice\}$. In principle no relation between the values of a symbolic feature is given. So we will assume that they are only different symbols[1].

Given an item space $X = \prod_{i=1}^{n} X_i$, we shall say that X contains the items $x^1, \ldots, x^N$, and $N = |X|$. Moreover, we shall denote with $x^j = (x_1^j, \ldots, x_n^j)$ the components of a vector $x^j \in X$.

B. Query Language

The query language that we are considering is quite simple but enough expressive to support typical form based queries to the catalogue. Let $X = \prod_{i=1}^{n} X_i$ be an item space, then a query Q is obtained by the conjunction of simple constraints, where each constraint involves only one feature. More formally, $Q = C_1 \wedge \cdots \wedge C_m$, where $m \leq n$, constraint C_k involves feature x_{i_k}, and:

$$C_k = \begin{cases} x_{i_k} = v_k & \text{if } X_{i_k} \text{ is symbolic} \\ l_k \leq x_{i_k} \leq u_k & \text{if } X_{i_k} \text{ is finite integer or real} \end{cases} \tag{1}$$

where $v_k \in X_{i_k}$, and l_k, u_k are two real numbers. A special value ALL (wild-card) can be used in a constraint on symbolic features meaning that the constraint is satisfied for all possible values.

Example. Let $X = AC \times Category \times Cost = \{\top, \bot\} \times \{1, 2, 3\} \times [0, 1]$, and $Q = (x_1 = \top) \wedge (20 \leq x_3 \leq 50)$. Then the query Q selects all the hotels that have air conditioning and room cost between 20 and 50. So if $X = \{(\top, 3, 60), (\top, 3, 50), (\bot, 3, 40), (\top, 2, 40)\}$, the the query will retrieve the items: $(\top, 3, 50), (\top, 2, 40)$.

[1]It is evident that relations can be defined in many cases, for instance, the "closeness" relation for the "Location" feature.

III. Query Relaxation

Query relaxation changes a query definition in such a way that the number of items returned by the query is increased. For instance $Q' = (20 \leq x_3 \leq 50)$ is a relaxation of the query $Q = (x_1 = \top) \wedge (20 \leq x_3 \leq 50)$, where the first constraint is relaxed. A query is typically relaxed when the result set retrieved from the item space X is void, or when the user is interested in having more examples to evaluate.

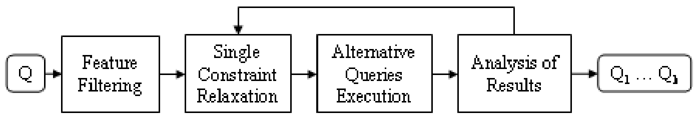

Fig. 2. The relaxation process.

The query relaxation suggestion process is depicted in Figure 2). In this situation IM takes as input a query $Q = C_1 \wedge \cdots \wedge C_m$ and builds a new set of relaxed queries $Q_1, \ldots Q_k$, such that each Q_i relaxes only one constraint of the original query. In the following discussion we shall use the query $Q = ((x_1 = \mathit{ski}) \wedge (x_2 = \mathit{cavalese}) \wedge (x_3 = \top) \wedge (x_4 \leq 100))$, where we are searching in a database of sport activities/services, and the features are $\mathit{SportType}, \mathit{Location}, \mathit{School}, \mathit{Cost}$. The relaxation process stages are the following:

1. **Features Filtering.** When a query must be relaxed, there are some feature constraints that cannot be changed without changing the information goal of the user. Referring to the example above, the user might well be interested in some schools a bit more expensive or maybe not in Cavalese but he is interested in skiing, not in fishing. This module identify those feature constraints that cannot be relaxed. In the example above it is found that only the *Cost*, *Location* and *School* features can be relaxed.
2. **Single Constraint Relaxation.** Whenever the relaxable constraints have been identified, for each of them a "relaxed" version (C'_i) must be identified. Two different approaches are exploited according to the feature type:
• **Symbolic feature constraint relaxation.** In this case relaxation means that the constraint is discarded.
• **Finite integer and real feature constraint relaxation.** In this case the range of allowed values is expanded by a certain percentage that varies according to the feature type.
After this stage the relaxed constraints in Q are: $x_2 = \mathit{ALL}$, $x_3 = \mathit{ALL}$ and $x_4 \leq 110$. The ALL keyword means that all the values are allowed for that feature.
3. **Alternative Queries Execution.** At this stage, for each "relaxable" constraint a new relaxed version of the original user's query is built, with only that constraint relaxed. In our example we have three new queries: $Q_1 = ((x_1 = \mathit{ski}) \wedge (x_2 = \mathit{ALL}) \wedge (x_3 = \top) \wedge (x_4 \leq 100))$, $Q_2 = ((x_1 = \mathit{ski}) \wedge (x_2 = \mathit{cavalese}) \wedge (x_3 = \mathit{ALL}) \wedge (x_4 \leq 100))$, and $Q_3 = ((x_1 = \mathit{ski}) \wedge (x_2 = \mathit{cavalese}) \wedge (x_3 = \top) \wedge (x_4 \leq 110))$. These relaxed queries are executed and the number of items retrieved by each single query is determined (counts)[2]
4. **Analysis of Results.** When the results (counts) are obtained they must be analysed. The goal is to understand if the relaxed queries have produced an improvement, i.e., if the new set of results (for each query) is not still void or some of them are now too large. Among those queries that still return a void (or too large) set two situations may arise: if the constraint relaxed involves a symbolic feature, then the query cannot be further relaxed; if the feature is integer or real then a new relaxation (or tightening) is tried by sending this information to the Single Constraint Relaxation module. Let us assume, for sake of simplicity, that, in our example: Q_1 returns 10 items, Q_2 returns none, and Q_3 returns 5 items. Then two new queries are suggested to the user Q_1 and Q_3, i.e., the systems tells to the user that there are ski schools: 1) that are not in Cavalese but cost less than 100; 2) or that are in Cavalese but cost a bit more but less than 110.

Some additional comments are in order. First, the list of features constraints that cannot be relaxed (in the Feature Filtering module) is defined statically for each query type in an application. In our prototype implementation a number of query types are considered and for each of them a meta-data lists those features that cannot be discarded in a query. Second, when relaxing a finite integer or real feature constraint IM uses again a meta-data information (for each feature type) to determine the percentage of increase in the range. For instance, when relaxing a "Cost" feature constraint, the user input range is increased by 10%.

A. The Relaxation Algorithm

We now describes the complete algorithm. The main algorithm is RelaxQuery(Q), it receives an input query to be relaxed and computes a set of relaxed queries. At the line 1 the output list of queries is initialized, then for each constraint in Q if it is relaxable in that query (line 3) the relaxation is tried. At line 4, the new relaxed query is initialized to null and the relaxation history (all the relaxation steps for that constraint) is initialized with the original constraint and the corresponding count that is set to 0. At line 8 the UpdateConstraint procedure is called, this must compute the relaxed or tightened form of that constraint according to the value of the relax flag (see below). If a true value is returned, then at line 9 a new query is produced. UpdateQuery(C, C', Q) returns a new query Q' in which C is replaced with C' in Q. The number of items satisfying the new relaxed query Q' are computed (Count(Q')).

Then Analyse(n, C', relax) is called. This procedure determines if a further relaxation (or tightening) step is needed. If a further relaxation is needed then at line 13 the current constraint and the corresponding count is appended to the relaxation history H, C is substituted with C', and a new iteration is initiated. If no further relaxation is required then the query and the count is appended to

[2]Actually the computation of these counters is faster than the complete execution of the query, that is the retrieval of items from the data base.

```
RelaxQuery($Q = C_1 \wedge \cdots \wedge C_m$)
Input: a query to be relaxed.
Output: a list $QL$ of relaxed queries.
1  $QL := \emptyset$
2  for $C$ in $Q$
3      if Relaxable($C, Q$)
4          $Q' := null, C' := null, H := \{(C, 0)\}$
5          $flag := true, relax := true, n := 0$
6          while $flag$
7              $flag := false$
8              if UpdateConstraint($C, C', H, relax$)
9                  $Q' :=$ UpdateQuery($C, C', Q$)
10                 $n :=$ Count($Q'$)
11                 $flag :=$ Analyse($n, C', relax$)
12                 if flag
13                     append $(C', n)$ to $H$
14                     $C := C'$
15         end while
16         $QL := QL \cup \{(Q', n)\}$
17 end for
18 return $QL$
```

Fig. 3. The query relaxation algorithm.

the result QL. We note that, it could still be the case that $n = 0$, i.e. even in the relaxed form there query still returns no items, or $n > \alpha$_threshold, that is, the relaxation has produced too many items (see description of the "Analyse" procedure for explanation of the α_threshold parameter). This is in any case a valuable information, and the client of the RelaxQuery procedure could decide how to better use this knowledge (e.g. whether to simply discard this relaxed query or notify the user about the failed attempt). The procedure exits from the while loop (line 6) when: 1) the UpdateConstraint procedure returns false (i.e., the maximum number of iterations is reached or no more improvement can be found according to the history; 2) the Analyse procedure returns false. The latter happens in the case the constraint C is symbolic or if $0 < n \leq \alpha$_threshold on a numeric feature (i.e., the number of items is satisfactory). Otherwise, the constraint C is on a numeric feature and $n = 0$ or $n > \alpha$_threshold, so a new attempt to produce a useful constraint will be tried.

We now describe the two additional procedures that are called in RelaxQuery. First we consider, UpdateConstraint that tries to relax a constraint or, if the previous relaxed version has obtained too many items, to tighten a bit the constraint (only for numeric features). The procedure is executed if the relaxation history length is less than a certain max_length value (this limits the number of times the UpdateConstraint procedure is called from RelaxQuery for a particular constraint). Then if the constraint is on a symbolic feature then it is completely deleted (line 6, $C' = (x = ALL)$, means that the constraint is always satisfied). Conversely, if it is on a numeric feature then the range constraint is either relaxed in line 7 (range increased by a percentage that depends on the feature) or tightened in line 9.

Second, the Analyse procedure, determines if a further relaxation step is required or not. If the type of C' is symbolic nothing is done (returns false). If the type of C is numeric and the count is still 0 or greater than the threshold α_threshold, then a further update (relaxation or tightening) is needed.

```
Analyse($n, C', relax$)
Input: count $n$, new constraint $C'$, "relax" variable.
Output: true if further refinement is needed, update "relax".
1  flag := false
2  if type($C'$) = numeric
3      if $n = 0$
4          $relax := true, flag := true$
5      if $n > \alpha$_threshold
6          $relax := false, flag := true$
7  return flag
```

Fig. 4. The analyses of results.

```
UpdateConstraint($C, C', H, relax$)
Input: constraint $C$ to be relaxed, relaxation history $H$,
relax or tighten flag "relax".
Output: return true if the constraint $C'$ is updated.
1  $result :=$ false
2  if length($H$) < max_length
3      if $relax$
5          if type($C$) = symbolic
6              $C' := (x = ALL), result := true$
7          else $result$:= ExpandRange($C, C', H$)
8              $result := true$
9      else if type($C$) = numeric
10         $result :=$ CutRange($C, C', H$)
11 return $result$
```

Fig. 5. The constraint relaxation algorithm.

IV. Query Tightening

The need for query tightening arises when a query returns too many items. By "too many" we mean exceeding a predefined threshold (we have called it α_threshold in the previous section). So, when the result size of a query exceeds the threshold, the system has to find some ways to refine the initial query or suggest to the user possible refinements such that it returns a smaller result set still satisfying the user needs.

Before we go into the details, we specify what we mean by query tightening. Let Q be a query constraining features $X_{i_1}, \ldots, X_{i_m}$, and assume Q has a large result size, then some new features must be found such that whenever one of those new features is constrained in Q, the number of items returned by Q is reduced.

The algorithm for tightening query is outlined in Figure 6. It takes as input the query Q and its result set. At line 2 the algorithm determines all the remaining features.

```
TightenQuery($Q = C_1 \wedge \cdots \wedge C_m, S_q$)
Input: $Q$ a query and $S_q$, the result set of $Q$
Output: At most 3 candidate features for further tightening the $Q$
1  $L := \emptyset$
2  Let $RF := \{X_{m+1}, \cdots, X_n\}$ be the set of all the remaining features
3  Let $SF$ be the set of all the selected features.
4  if $|S_q| \geq \beta$_threshold then
5      for $i = m+1$ to $n$ do
6          $\lambda := H_S(X_i)$
7          append $(\lambda, X_i)$ pair to L
8      end for
9  else
10     for $i = m+1$ to $n$ do
11         $\lambda := \frac{H_{S_q}(X_i)}{max\{H_{S_q}(X_i; X_j) | X_j \in SF\}}$
12         append $(\lambda, X_i)$ pair to L
13     end for
14 end if
15 return the top 3 scored features from $L$
```

Fig. 6. The query tightening algorithm.

Without loss of generality, for the sake of simplicity we assume the first m features are constrained in Q. When the cardinality of the result set has reached a predefined threshold, say β_threshold, the entropy [8] function is used to score each feature in RF. It measures the amount of information contained in a feature. The entropy of a feature is computed as follows

$$H_A(X_i) = -\sum_{v \in X_i} p(v) \log[p(v)], \quad X_i \in RF \tag{2}$$

where $p(v)$ is the likelihood of occurring the value v of feature X_i in the set $A \subset X$. Hence,

$$p(v) = \frac{\text{\# of items in } A \text{ containing } v}{|A|}$$

All the entropies $H_S(X_i)$ $(1 \leq i \leq n)$ are pre-computed over the whole data set S (the catalogue), and hence no on-line computation is needed in this case.

The above definition of entropy applies when features are symbolic. For numeric features, the values have to be discretized, for instance in equi-depth buckets [9]. Then the same formulas can be applied as well. In case the β_threshold is not reached, the remaining features are scored using entropy taking into account its relation with already selected features using mutual information [8]. At line 11, all the entropies and mutual information are computed on the result set of query denoted by S_q. Here $H_{S_q}(X_i; X_j)$ is the mutual information between X_i and X_j. It is defined as

$$H_A(X_i; X_j) = H_A(X_i) + \sum_{v \in X_i,\, u \in X_j} p(v, u) \log[p(v|u)] \tag{3}$$

where $A \subset X$ and

$$p(v, u) = \frac{\text{\# of items in } A \text{ containing } v \text{ and } u}{|A|}$$

and

$$p(v|u) = \frac{\text{\# of items in } A \text{ containing } v \text{ and } u}{\text{\# of items in } A \text{ containing } u}$$

In analogous way we can compute the mutual information between numeric features or between symbolic and numeric features.

The mutual information between two features measures the amount of information that feature X_i conveys about feature X_j, and it is high (low) when features are strongly (less) correlated. Selecting a feature that is strongly related to an already presented feature in the query, may not have much impact on the result set, and hence no improvement. This is the reason the maximum value is taken at line 11, thus the λ value is high when the feature X_i gives high information (entropy is high) and it is less related to all X_j selected features.

After the scores (i.e, the λ values) of all remaining features have been computed, at most the 3 highest scored features are returned by the algorithm.

V. Discussion and Related Work

The above presented procedures to relax or tightening user queries are still limited in a number of ways. We now briefly mention a set of improvements that are under development.

First of all, we are implementing methods that enable the Feature Filtering module to exploit session information for determining those constraints that cannot be relaxed. For instance, if the user has already selected a destination, then the "Location" feature in a service search is not proposed for relaxation.

Secondly, the constraints on symbolic features should not completely be discarded in the Single Constraint Relaxation module. Using a type hierarchy, the equality constraint can be relaxed going up in the hierarchy or substituted with a range constraint. For instance if the constraint is *Location = "cavalese"* this may be relaxed to *Location = cavalese ∨ molina* or even better, introducing specific relaxation operators, e.g. *Location = Close(cavalese).*

Thirdly, the method used to generate relaxed queries is very simple, i.e., only one constraint is changed and this is repeated for all those constraints that can be relaxed. This choice is intended to limit the combinatorial explosion of such changes. In fact if a query has m constraints the number of relaxed queries that can be generated modifying an arbitrary number of constraints is 2^m. Hence a method for limiting the nature of the changes is to be adapted. We are implementing a refined version that limits the number of relaxed queries proposed by the system but can propose relaxations that simultaneously operates on more that one feature.

Fourthly, the two feature selection methods used at lines 6 and 11 of algorithm TightenQuery has been introduced as heuristics to search for the best features for reducing the result size of a query. In the futures, we shall evaluate these methods together with other possible methods (e.g., using mutual information alone) using empiric result to see how they behave in the real application.

Related work. The idea of extending a mediator system with support for query refinement is quite new, but fits the goal of cooperative database research, that aims to create information systems that attempt to understand the gist of a user query, to suggest or answer related questions, to infer an answer from data that is accessible, or to give an approximate response [10]. An approach similar to that presented in this paper has been investigated in the CoBase system (www.cobase.cs.ucla.edu) [11], [12]. Here, a relaxation mediator provides operations such as approximately-to, similar-to, or near-to on specific data schemas and/or types. In addition, a relaxation mediator searches for approximate answers automatically whenever a user query is over or ill specified. The relaxation mediators use a knowledge structure termed Type Abstraction Hierarchy (TAH) to assist in approximately answering queries. TAH is used to represent objects at different level of abstraction and hence to produce relaxed versions of a constraint on a single feature.

Among the major differences between our approach and CoBase, we must note that Cobase is not able to suggest new features for further tightening, it is mostly concerned with relaxation. Besides, IM does not use a TAH to determine the alternative single feature constraint but exploits a simpler method based on the type of the constrained feature (symbolic vs. numeric) plus additional knowledge stored in a metadata component. Moreover we must note that the control of the relaxation (search for the right relaxation) is not fully described in the CoBase literature and therefore is impossible to compare our search algorithms (RelaxQuery) with their.

The idea of supporting user in the refinement of a query is also inspired by interactive data exploration, that is popular in OLAP and Data Warehouse systems [13]. Here, complex query languages are offered to the user, so that he can navigate through the data space. Even if these technologies can be used to implement the relaxation and tightening functions offered by IM these are not immediately available in OLAP systems.

VI. Conclusions

This paper describes a work in progress at a newly established research center on eCommerce and Tourism (http://ectrl.itc.it).

The current result of our work is a methodology to ease the selection of items from a catalogue by suggesting relevant changes to the users' queries, avoiding a typical problem that occurs in these situations: the query constraints are too strict and no suggestion can be made. More in general, this is only a component of a comprehensive middleware for issuing personalized recommendation to a leisure traveler in identifying and aggregating elementary services or activities to be consumed. It is based on the principle that effective suggestions are based on a combination of factors like: appropriate destination modeling, data retrieval and filtering with both precise and approximate matching, scoring using personal preferences that can be derived from a base of previous cases [14], [4].

The proposed approach has been implemented for the "Azienda di Promozione Turistica" of Trentino (Trentino Destination Management Organization), and will be shortly validated within a use group. A more comprehensive version will be developed as main result of an European IST Project (DIETORECS, in collaboration with: TIScover AG - Travel Information Systems (A), Institute for Tourism and Leisure Studies - Vienna University of Economics and Business Administration (A), National Laboratory for Tourism and eCommerce - University of Illinois at Urbana-Champaign (USA) and Azienda per la Promozione Turistica del Trentino (I)).

Acknowledgements

This work has been partially funded by CARITRO foundation (under contract "eCommerce e Turismo") and by the European Union's Fifth RTD Framework Programme (under contract DIETORECS IST-2000-29474).

References

[1] Paul Resnick and Hal R. Varian, "Recommender systems," *Communications of the ACM*, vol. 40, no. 3, pp. 56–58, 1997.

[2] J. Ben Schafer, Joseph A. Konstan, and John Riedl, "E-commerce recommendation applications," *Data Mining and Knowledge Discovery*, vol. 5, no. 1/2, pp. 115–153, 2001.

[3] Robin Burke, *Encyclopedia of Library and Information Science*, vol. 69, chapter Knowledge-based Recommender Systems, Marcel Dekker, 2000.

[4] Francesco Ricci and Hannes Werthner, "Case-based querying for travel planning recommendation," *Information Technology and Tourism*, vol. 4, no. 3/4, pp. 215–226, 2002.

[5] Gio Wiederhold, "Mediators in the architecture of future information systems," *IEEE Computer*, vol. 25, no. 3, pp. 38–49, 1992.

[6] Daniela Florescu, Alon Levy, and Alberto Mendelzon, "Database techniques for the world-wide web:a survey," *SIGMOD Record*, vol. 27, no. 3, pp. 59–74, 1998.

[7] Francesco Ricci, Nader Mirzadeh, Adriano Venturini, and Hannes Werthner, "Case-based reasoning and legacy data reuse for web-based recommendation architectures," in *Proceedings of the Third International Conference on Information Integration and Web-based Applications & Services*, Linz, Austria, September 10-12 2001, pp. 229–241.

[8] Thomas M. Cover and Joy A. Thomas, *Elements of Information Theory*, John Wiley & Sons, New York, NY, USA, 1991.

[9] Ian H. Witten and Eibe Frank, *Data Mining*, Morgan Kaufmann Publisher, 2000.

[10] Terry Gaasterland, Parke Godfrey, and Jack Minker, "An overview of cooperative answering," *Journal of Intelligent Information Systems*, vol. 1, no. 2, pp. 123–157, 1992.

[11] Wesley W. Chu, Hua Yang, Kuorong Chiang, Michael Minock, Gladys Chow, and Chris Larson, "Cobase: A scalable and extensible cooperative information system," *Journal of Intelligence Information Systems*, vol. 6, 1996.

[12] Wesley W. Chu, Hua Yang, and Gladys Chow, "A cooperative database system (cobase) for query relaxation," in *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems*, Edinburgh, May 1996.

[13] Surajit Chaudhuri and Umeshwar Dayal, "An overview of data warehousing and olap technology," *SIGMOD Record*, vol. 26, no. 1, pp. 65–74, 1997.

[14] Agnar Aamodt and Enric Plaza, "Case-based reasoning: foundational issues, methodological variations, and system approaches," *AI Communications*, vol. 7, no. 1, pp. 39–59, 1994.