Techniques for Fast Query Relaxation in Content-Based Recommender Systems

Dietmar Jannach

Institute for Business Informatics & Application Systems
University Klagenfurt
dietmar.jannach@uni-klu.ac.at

Abstract. 'Query relaxation' is one of the basic approaches to deal with unfulfillable or conflicting customer requirements in content-based recommender systems: When no product in the catalog exactly matches the customer requirements, the idea is to retrieve those products that fulfill as many of the requirements as possible by removing (relaxing) parts of the original query to the catalog. In general, searching for such an 'maximum succeeding subquery' is a non-trivial task because a) the theoretical search space exponentially grows with the number of the subqueries and b) the allowed response times are strictly limited in interactive recommender applications.

In this paper, we describe new techniques for the fast computation of 'user-optimal' query relaxations: First, we show how the number of required database queries for determining an optimal relaxation can be limited to the number of given subqueries by evaluating the subqueries individually. Next, it is described how the problem of finding relaxations returning '*at-least-n*' products can be efficiently solved by analyzing these partial query results in memory. Finally, we show how a general-purpose conflict detection algorithm can be applied for determining 'preferred' conflicts in interactive relaxation scenarios.

The described algorithms have been implemented and evaluated in a knowledge-based recommender framework; the paper comprises a discussion of implementation details, experiences, and experimental results.

1 Introduction

Content-based recommendation approaches employ detailed knowledge about the items in the product catalog. In addition, in particular in interactive and knowledge-based recommender systems, the customer's requirements are in many cases directly or indirectly mapped to product characteristics, which also means that the set of suitable products is determined by dynamically constructing a query to the catalog or database [9]. One of the main problems of such filter-based retrieval methods, however, is that situations can easily arise in which none of the products fulfills all of the customer requirements [2]. 'Query relaxation', i.e., the removal of parts of the query, is beside similarity-based retrieval one of the commonly used approaches to deal with this problem: The main goal of such approaches is to retrieve products that fulfill as many of the customer's

C. Freksa, M. Kohlhase, and K. Schill (Eds.): KI 2006, LNAI 4314, pp. 49–63, 2007.

requirements as possible - a task that can be mapped to the problem of finding a maximum succeeding subquery (XSS) [3] of the original query.

Recently, new algorithms for determining such XSS and Minimally Failing Subqueries (MFS) [3] have been proposed in the context of recommender systems, see e.g., [9,10,12] and it has been shown that query relaxation can be a helpful technique for implementing more intelligent behavior in recommender systems, e.g., for building an interactive relaxation facility or for the generation of explanations for the proposals.

The main problem in finding suitable relaxations, however, lies in the fact that the theoretical search space exponentially increases with the number of atoms (subqueries) of the original query, i.e., if the query can be split into n subqueries there exists 2^n possible combinations.[1] At the same time, the allowed time frame for the system's response is very short in interactive applications, i.e., response times should be below one second: McSherry [9] addresses this problem by precomputing and filtering the set of MFSs by cardinality, which, however, still requires a significant number of database queries. Ricci [12] introduces *Feature Abstraction Hierarchies* for coping with the complexity problem, which results in short response times at the cost of incompleteness. In addition, both approaches primarily rely on the assumption that smaller relaxations (in terms of cardinality) are always preferable, an assumption that might not be true in all application domains.

In this paper, we propose new techniques for determining optimal or preferred relaxations in an efficient way and which help us to overcome the limitations of previous approaches: For the *non-interactive* case (Section 2), in which the system immediately computes a relaxation when the query fails, we propose to evaluate the atoms of the query individually in a preprocessing step and base the subsequent computation of relaxations on these intermediate results. Furthermore, we also show how we can efficiently determine relaxations that lead to *at least n* items, because in some application domains it is desirable that the recommendation comprises more than one single product, such that the end user has a choice of multiple products that he can compare. This computation is again based on the partial query results and is a form of relaxation which is not covered by previous approaches.

For supporting *interactive* relaxation (Section 3), i.e., scenarios in which the end user incrementally states on which requirements s/he is willing to compromise, we show how a recent *conflict-detection* algorithm can be utilized for fast computation of *preferred* conflicts. In contrast to previous approaches, in our approach the conflicts are computed *on demand*, which means that the costly process of finding all conflicts in the requirements in advance (like in [9]) is not required.

In Section 4 finally, we discuss implementation aspects and summarize the experiences gained from several real-world advisory applications in different domains. The paper ends with a short discussion of further and other related work.

[1] A detailed complexity analysis for the MFS/XSS problem is given in [3].

2 Non-interactive Relaxation

The basic problem of query relaxation lies both in the size of the search space and in the limited response times: The theoretical search space for the case $n=4$ is illustrated in Figure 1. In fact, in previous approaches [9,10,12] the search for relaxations is based on scanning this lattice; the lattice also served as a basis for the complexity analysis in [3].

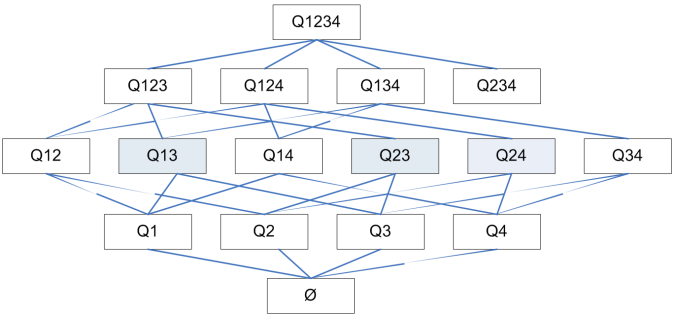

Fig. 1. Lattice of possible subqueries, minmal relaxations are printed in shaded boxes

In contrast to these approaches, we base our first two algorithms on the concept of *product-specific relaxations (PSX)* and the individual evaluation of the atoms of the query: A *PSX* for a product p in the catalog corresponds to the set of atoms of the original query that filters out p from the result set. These sets can be efficiently computed in-memory based on the partial query results, which shall be demonstrated in the following example. Figure 2 depicts a typical product catalog from the domain of digital cameras.

ID	USB	Firewire	Price	Resolution	Make
p1	true	false	400	5 MP	Canon
p2	false	true	500	5 MP	Canon
p3	true	false	200	4 MP	Fuji
p4	false	true	400	5 MP	HP

Fig. 2. Product database of digital cameras

Let the user's query consist of the following requirements (atoms) which results in a theoretical search space of $2^4 = 16$ combinations of atoms.

$Q = \{$ usb = true (Q1), firewire = true (Q2), price < 300 (Q3),
resolution >= 5 MP (Q4)$\}$

Given the catalog and the user query, we can compute a matrix of zeros and ones (see Figure 3) that shows which atoms of the query filter out which products. Note that for constructing this matrix, we need exactly four database queries; when using bit-set data structures, we only need *nbProducts*nbAtoms* bits for storing this data in memory. In addition, determining the matching set of products for the overall query can be efficiently done by using fast *bitwise-and* operations on the rows of the matrix.

ID	p1	p2	p3	p4
Q1	1	0	1	0
Q2	0	1	0	1
Q3	0	0	1	0
Q4	1	1	0	1

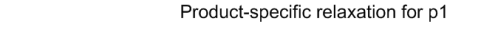

Fig. 3. Evaluating the subqueries individually

For determining a *PSX* for a given product p_i we can look up in the matrix the fields that have a zero in the corresponding column which directly leads to the set of atoms we would have to relax in order to have p_i in the result set. In the example, the list of $PSXs = \langle\{Q2, Q3\}, \{Q1, Q3\}, \{Q2, Q4\}, \{Q1, Q3\}\rangle$ and the set of minimal relaxations is $\{\{Q1, Q3\}, \{Q2, Q3\}, \{Q2, Q4\}\}$: Considering, for example, $\{\{Q1, Q3\}$ as a relaxation leading to $p1$, we see that when relaxing only $Q1$ or $Q3$ alone, no product will be in the result set (see Figure 1). In general, however, not all PSX's are already *minimal* relaxations: If, for instance, there is a camera *p5* with neither USB nor Firewire support at a price of 400, the PSX for *p5* would be $\{Q1, Q2, Q3\}$, which would be a non-minimal relaxation of the problem because it is a superset of another PSX. However, we will subsequently show that the set of *minimal* relaxations always is among the list of product-specific relaxations of the problem and we can determine the minimal relaxations by scanning this list. In addition, we can also easily rank the different relaxations, when we are given a cost function that associates relaxation costs with each part of the query. We base our definitions on the work of [3] and [9], respectively.

Definition 1. *(Query): A query Q is a conjunctive query formula, i.e., $Q \equiv A_1 \wedge ... \wedge A_k$. Each A_i is an atom (condition).*

In the following we denote the number of atoms of the query as $|Q|$ (query length).

Definition 2. *(Subquery): Given a query Q consisting of the atoms $A_1 \wedge ... \wedge A_k$, a query Q' is called a subquery of Q iff $Q' \equiv A_{s_1} \wedge ... \wedge A_{s_j}$, and $\{s_1, ...s_j\} \subset \{1, ..., k\}$*

Lemma 1. *If Q' is a subquery of Q and Q' fails, also the query Q itself must fail.*

We now define the term *relaxation* which is more common in the application domain than *Maximal Succeeding Subqueries* in the sense of [3]. Of course, these things are directly related to each other.[2]

Definition 3. *(Valid relaxation): If Q is a failing query and Q' is a succeeding subquery of Q, the set of atoms of Q which are not part of Q' is called a valid relaxation of Q.*

Definition 4. *(Minimal relaxation): A valid relaxation R of a failing query Q is called minimal, if there exists no other valid relaxation R' of Q which is a subset of R.*

In Figure 1, the minimal relaxations for the example problem are depicted in shaded boxes.

Definition 5. *(Maximal succeeding subquery - XSS): Given a failing query Q, a Maximal Succeeding Subquery XSS for Q is a non-failing subquery of Q and there exists no other query Q' which is also a non-failing subquery of Q for which holds that XSS is a subquery of Q'.*

Lemma 2. *Given a maximal succeeding subquery XSS for Q, the set of atoms of Q which are not in XSS represent a minimal relaxation R for Q.*

Lemma 3. *If the product catalog P is not empty, a relaxation R for Q will always exist.*

A product-specific relaxation can be defined as follows:

Definition 6. *(Product-specific relaxation - PSX): Let Q be a query consisting of the atoms A_1, ..., A_k, P the product catalog, and p_i an element of P. $PSX(Q, p_i)$ is defined to be a function that returns the set of atoms A_i from $A_1, ..., A_k$ that are not satisfied by product p_i.*

Lemma 4. *The set of atoms returned by $PSX(Q, p_i)$ is also a valid relaxation for Q.*

We have to show that all minimal relaxations are contained in the PSXs of the products.

Proposition 1. *Given a failing query Q and a product database P containing n products, at most n minimal relaxations can exist and all minimal relaxations are among the PSXs for Q and P.*

[2] The term 'query relaxation' is also used in the context of XML databases, where the goal is to find *approximate* answers to user queries. These approaches, however, have little relation with our work and mainly aim at relaxing structural constraints in general XML-specific query languages [1,7].

ALGORITHM: *MinRelax*
In: A query Q, a product catalog P
Out: Set of minimal relaxations *minRS* for Q

```
MinRS = ∅
forall p_i ∈ P do
    PSX = Compute the product-specific relaxation PSX(Q, p_i)
    % Check relaxations that were already found
    SUB = {r ∈ MinRS | r ⊂ PSX}
    if SUB ≠ ∅
        % Current relaxation is superset of existing
        continue with next p_i
    endif
    SUPER = {r ∈ MinRS | PSX ⊂ r}
    if SUPER ≠ ∅
        % Remove supersets
        MinRS = MinRS \ SUPER
    endif
    % Store the new relaxation
    MinRS = MinRS ∪ {PSX}
endfor
return MinRS
```

Fig. 4. Algorithm for determining all minimal relaxations

Proof. For each product $p_i \in P$ there exists exactly one subset PSX of atoms of Q which p_i does not fulfill and which have to be definitely relaxed altogether in order to have p_i in the result set. Given n products in P, there exist exactly n such PSXs. Thus, any valid relaxation of Q has to contain all the elements of at least one of these PSXs for obtaining one of the products of P in the result set. Consequently, any relaxation which is not in the set of all PSXs of Q has to be a superset of one of the PSXs and is consequently no longer a minimal relaxation. This finally means that any minimal relaxation must be contained in the PSXs of all products and not more then $|PSX| = n$ such minimal relaxations can exist.

Computing the optimal relaxation. The computation of the optimal relaxation (in terms of relaxation costs) can be done by a simple scan of the *PSXs* of the relaxation problem: Given an arbitrary cost function that takes for instance the cardinality of the relaxation and/or individual costs for the individual atoms into account, we only have to determine the *PSX* that minimizes that function. The only assumption for that is that the costs for a superset of a given *PSX* must not be lower than the costs of that *PSX* itself.[3]

[3] Within the ADVISOR SUITE system (see later), the 'costs' for relaxing a certain subquery are defined a-priori by a domain expert. Other forms of acquiring the cost function, e.g., by analyzing the user behavior, are also possible.

Computing all minimal relaxations. The set of all minimal relaxations (comparable to the *Recovery Set* from [10]) can be computed with the help of Algorithm *MinRelax* by removing supersets from the set of all PSXs.

Proposition 2. *Algorithm MinRelax is sound and complete, i.e., it returns exactly all minimal relaxations for a failing query Q.*

Proof. The algorithm iteratively processes the product-specific relaxations PSXs for all products $p_i \in P$. From Lemma 4 we know that all these PSXs are already valid relaxations. Minimality of the relaxations returned by $MinRelax$ is guaranteed by the algorithm, because a) supersets of already discovered PSXs are ignored during result construction and b) already discovered PSXs that are supersets of the current PSX are removed from the result set. As such, there cannot exist two relaxations $R1$ and $R2$ in the result set for which $R1$ is a subset of $R2$ or vice versa. In addition, we know from Proposition 1 that all minimal relaxations are contained in the PSXs of the products of P. Since $MinRelax$ always processes all of these elements, it is guaranteed that none of the minimal relaxations is missed by the algorithm.

Finding relaxations with at least *n* products. In practical applications, it is sometimes not desirable just to present one single product to the user but rather have at least a few products in the proposal that could for instance be used for comparison purposes. Note that the relaxations computed with the algorithms described above only guarantee that at least one product will be in the result set. In the following paragraphs we thus show how we can compute such relaxations that have *at least n* products based on our in-memory data structures without the need for further database queries.

We use the example shown in Figure 5 (with seven products and four subqueries) for illustrating a corresponding algorithm for determining such relaxations. In this example, the list of product-specific relaxations is as follows:
$PSXs = \langle$ {f4}, {f1, f4}, {f3}, {f2, f3}, {f1}, {f4}, {f1, f3} $\rangle$

ID	p1	p2	p3	p4	p5	p6	p7
f1	1	0	1	1	0	1	0
f2	1	1	1	0	1	1	1
f3	1	1	0	0	1	1	0
f4	0	0	1	1	1	0	1

Fig. 5. Problem setting for n products

Algorithm *NRelax* (Figures 5 and 6) computes relaxations that lead to at least n products based on the list of *PSXs*, meaning that no further database queries are required. Note that *NRelax* starts with the full list of the original *PSXs*, because using only the minimal relaxations would be not sufficient for our purposes and the optimal relaxation for 'at least n' products could be lost.

The algorithm works by incrementally exploring combinations of the individual *PSXs*: The algorithm starts with the initial list of *PSXs* and systematically constructs the possible combinations in order of their cardinality. When two *PSXs* are to be combined, the union of the involved atoms is generated, the number of products for the relaxation is determined, and the corresponding cost function is calculated. When a combination is found that leads to enough products, we remember the cost value and subsequently prune the search space by removing combinations that cannot lead to a better result anymore.

The following example shall illustrate the details of the algorithm. We use a simple cost function, i.e., relaxing the filter condition f_n shall lead to costs of n.

In the first step, we remove the duplicates from the list of *PSXs* and annotate each element with the corresponding costs and the number of products that will result in that relaxation and sort the elements according to the costs. Determining the number of products for a certain relaxation can be done by checking for subset relations in *PSXs*, e.g., $\{f1, f3\}$, will have costs of "4" and will result to 3 products, since $\{f1\}$ and $\{f3\}$ are also elements of the list of *PSXs*. The collapsed list of *PSXs* named *CPSX* for our example therefore is the following, where $\{f3\}(3/1)$ denotes that the relaxation *f3* has costs of 'three' and results in one product.

$CPSX = \langle$ {f1} (1/1), {f3} (3/1), {f4} (4/2), {f1,f3} (4/3), {f1, f4} (5/4), {f2, f3} (5/2) $\rangle$

Starting with this initial list, we now compute the combinations of the elements of *CPSX* and use the data structure *RNode* for storing costs and number of products associated with an element in $CPSX$, i.e.,

```
struct RNode:
    atoms: List of atoms of PSX
    cost: costs of node
    nbProducts: number of products
    closed: flag, if node was closed
endstruct
```

and use an 'agenda' (list) of such nodes to remember the combinations that still have to be explored.

Figure 6 illustrates how the combinations are generated and how the search space can be pruned. The different aspects of the algorithm are marked with numbers in the Figure: At (1), a new node {f1,f3} is constructed from {f1} and {f3} respectively. At (2), the node {f1,f3} from the current agenda (at the first level in Figure 6) can be closed as it will be further explored in the next round of expansion. In Figure 5 this fact is indicated with an 'x'; At (3), the successor of {f1} and {f1,f3} would be {f1,f3}. However, we already found that node in (1) and can ignore it for further exploration; in Figure 5, nodes that are pruned from the search space in that way are marked with a rhombus. At (4) we have found a relaxation that comprises all possible atoms, which means that we do not explore that node any further.

Note that the number of nodes on the top level (and more importantly, overall) is limited to the number of possible relaxations for the given query, i.e. if

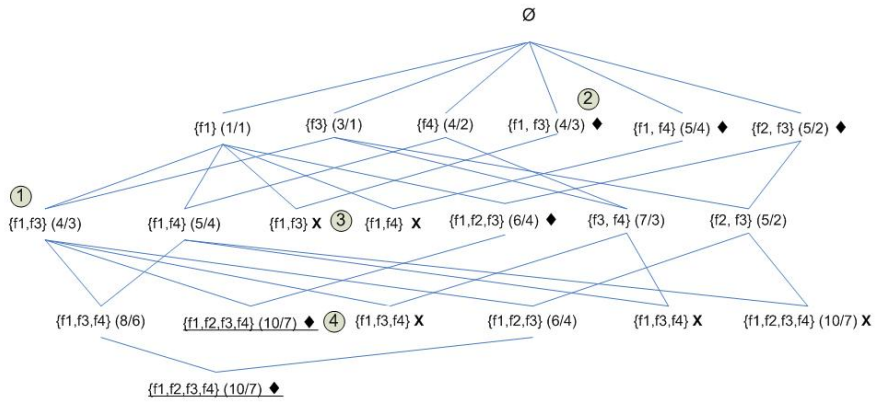

Fig. 6. Searching for at least n products

$|query| = n$, there can only be $2^n - 1$ nodes in the worst case, independent of the number of products in the catalog. Still, if there are already $2^n - 1$ different nodes on the first level, no further expansion will be required since all possible combinations are already contained in this first level.

All the computations can be done on the basis of the pre-evaluated partial results and do not require any further database queries. Also, compared with an approach that works by constructing all 2^n combinations of all possible atoms of the original query, we can also restrict the search space based on the already existing partial results and can leave out those that will definitely lead to more products: In our example, {f2} is not a product-specific relaxation and we therefore will never consider useless combinations like {f1,f2}, {f2,f4} and so forth, since we already now that no additional product will be in the result set when adding {f2} alone. Still, the completeness of the algorithm is still guaranteed by the systematic construction of all possible combinations of the *PSXs*.

Finally, in practical and more complex examples, the described cost-based tree pruning techniques will significantly reduce the number of nodes to be explored, which is not shown in the example i.e., only small fractions of the theoretical search space will be explored. If we, for instance, search for a relaxation with at least 3 products for the given example, we will find {f1,f3} to be the best relaxation in the original agenda and will not have to add a second-level element to the agenda due to cost-optimality of the node.

3 Interactive Relaxation

The alternative approach to immediately computing a relaxation is to let the user decide interactively on which requirements (s)he is willing to compromise. [9] proposes a corresponding algorithm, which is based on the concept of Minimally Failing Subqueries:

```
ALGORITHM: NRelax
In: A set of product-specific relaxations PSXs, threshold n
Out: An optimal relaxation for n products
CPSX = Collapse and sort PSXs as sequence of RNodes
bestNode = new RNode with infinite costs.
return NRelaxInt(CPSX,n, bestNode)

function NRelaxInt(agenda,n, bestNode)
In: agenda: Sequence of RNodes to explore, n: threshold,
    bestNode: currently best node
Out: An optimal relaxation for n products
if |agenda| = 0 return bestNode
% Check for new optimum
newBest = node from agenda with lowest costs for which node.costs
     are lower than bestNode.costs and nbProducts > n
if newBest ≠ null then bestNode = newBest endif
% Set up new agenda
newAgenda = <>
% Combine elements of given agenda
for i=0 to |agenda| − 1
     for j=i+1 to |agenda|
      n1 = agenda[i]
      n2 = agenda[j]
      % ignore closed nodes
      if n1.closed or n2.closed continue with next j endif
      newNode = combine atoms, costs, products of n1 and n2
      if not exists n ∈ newAgenda where n.atoms = newNode.atoms
       Close nodes n in agenda, for which n.atoms=newNode.atoms
       if newNode.costs < bestNode.costs and
         newNode does not contain all possible atoms
           add newNode to newAgenda endif
      endif
    endfor
endfor
NRelaxInt(newAgenda,n, bestNode)
return bestNode.atoms
```

Fig. 7. Algorithm for finding relaxation with n products

Definition 7. *(Minimal Failing Subquery - MFS): A failing subquery Q^* of a given query Q is a minimally failing subquery of Q if no proper subquery of Q^* is a failing query.*

As an alternative to McSherry's approach which relies on the possibly costly computation of *all* MFSs before starting the interactive relaxation process, we propose to apply Junker's recent QUICKXPLAIN [6] algorithm for computing MFSs *on demand*: The overall scenario is that when we have the situation of unfulfillable user requirements, we aim at finding a preferable and minimal con-

flict in these requirements and let the user decide how to proceed. *Preferred* means that we shall try to identify those conflicts (among possibly many conflicts) that contain requirements for which we assume that a typical user might be willing to compromise. In the digital camera domain we could for instance assume or learn that experts in digital photography searching for cameras supporting 'firewire' connectivity are rather willing to compromise on the price than on the technical requirement.

In general, the required priority values for each requirement can either be annotated in advance or they can be learned from the interaction history of different users over time. The implementation and evaluation of such a module that dynamically adapts priorities over time is part of our ongoing work. Originally, QuickXPlain was developed for finding conflicts (corresponding to MFSs) in Constraint Satisfaction problems, but its general, non-intrusive nature allows us to adapt it for our purposes (Figure 8). QuickXPlain is based on a *divide-and-conquer* strategy: In the decomposition phase it partitions the problem into subproblems of smaller size (thus pruning irrelevant parts of the problem) and subsequently tries to re-add individual elements and checks for consistency while at the same time taking preferences into account. Depending on the number of atoms in the query n, the size of the preferred conflict k, and the splitting point (e.g., *n/2*), QuickXPlain in the best case only needs $log(n/k)+2k$ consistency checks (database queries in our case) and $2k * log(n/k) + 2k$ in the worst case [6]. When we consider our initial example from Section 2, we see that there are three minimal conflicts in the requirements, i.e., {*usb = true, firewire = true*}, {*firewire = true, price* $\leq$ *300*}, {*price* $\leq$ *300, resolution* $\geq$ *5MP*}. Let us assume that the partial order $\prec$ (in the sense of [6]) among the attributes in the requirements is "*price* $\prec$ *firewire* $\prec$ *usb* $\prec$ *resolution*". Given these priorities, our adapted QuickXPlain (Figure 8) will immediately split the atoms (denoted as P,F,U and R for short) of the query into {P,F} and {U,R}. In the first recursive call, the algorithm will detect that {P,F} contains conflicting requirements and thus proceeds by further analyzing this subset alone, which means that half of the atoms in that example do not have be taken into consideration in subsequent steps. Next, QuickXPlain proceeds with the next sets of atoms {P} and {F} in our example can immediately determine that both {P} and {F} have to be in the minimal conflict, since the number of the remaining atoms $|A| = 1$ in both cases. Thus, the algorithm returns the preferred conflict {P,F}, because given the priorities in the example, it is preferable for the user to give up the price or the firewire requirement than to give up the requirement on the desired resolution.

A general algorithm that shows how QuickXPlain can be integrated into an interactive relaxation procedure is sketched in Figure 9. Please note that with the help of conflicts computed with QuickXPlain we can also compute the set of minimal or optimal relaxations (or XSSs) based on Reiter's [11] Hitting-Set Algorithm (see also [5]): This can be seen as an alternative to our approach based on product-specific relaxations from the previous section. The run-time performance of such an adapted Hitting-Set algorithm has been evaluated in [5] for different problem instances: The results showed that even if we do not rely

ALGORITHM: *mfsQX*
In: A failing query Q
Out: A preferred conflict of Q

A = *sorted list of atoms of* Q
return $mfsQI(\emptyset, A)$

function *mfsQI* (BG, A)
In: BG: List of atoms in background
A: List of atoms of failing query
Out: A preferred conflict of Q

% Construct and check the current set of atoms
$query = \bigwedge_{b \in BG}(b)$
if *query* is not successful
return $\emptyset$
endif
if $|A| = 1$
return A
endif
% Split remaining atoms into two parts
$C1 = \{a_i \in A | i < (|A|/2)\}$
$C2 = A \setminus C1$
% Evaluate branches
$\Delta_1 = mfsQI(BG \cup C1, C2)$
$\Delta_2 = mfsQI(BG \cup \Delta_1, C1)$
return $\Delta_1 \cup \Delta_2$

Fig. 8. Using QuickXPlain for computing preferred MFS

on the pre-computation of the *PSXs* no more than one second is required for finding the optimal relaxation also for the hard instances.

4 Implementation and Evaluation

All of the described algorithms and techniques have been implemented within the Java-based ADVISOR SUITE system [4], a knowledge-based framework for the rapid development of interactive advisory systems. Within that system, product retrieval is initially[4] based on if-then-style *filtering rules*, like *'If the user is interested in high-connectivity, propose products that support firewire'* or *'Only propose products that are cheaper than the customer-specified limit'*. Note that the consequent of the rules can contain arbitrary boolean formulae, i.e., also disjunctions. The rules are maintained by the domain expert or knowledge engineer with the help of graphical tools and can also be annotated with corresponding relaxation costs and explanatory texts, both for the case that the rule could be applied and for the case of relaxation (see Figure 10).

[4] After the initial determination of suitable products, these products are sorted based on a utility-based approach.

```
ALGORITHM: InteractiveRelax
In: Sorted list of atoms A of failing query Q
query = ⋀_{a∈A}(a)
if query is not successful
    % Compute a minimal preferred conflict
    conflict = mfsQX(∅, A)
    remaining = conflict
    % Set up the choice points
    do |conflict| times
        choice = Ask user to select an option from
                 remaining or 'backtrack'
        if choice = 'backtrack' return
        remaining = remaining \ {choice}}
        % Remove the choice and try again
        interactiveRelax(A \ {choice})
    end do
else
    Minimize the relaxation and compute results
    Report success and show proposal to user.
    response = Ask user if happy with result
    if response = 'yes'
        exit function
    % backtrack to last choice point
    else return
```

Fig. 9. Basic algorithm for interactive relaxation

We tested our algorithms with several real-world knowledge-bases from different domains and with different complexities; an average case would be a setting where we have 10-15 atoms (filter rules) in the query to be relaxed and a few hundred products in the catalog. All of the relaxation problems (see [5] for more details on running times) could be easily solved within the targeted time frame of one second, most of them much faster: Remember that for computing the user-optimal relaxation we only need $|query|$ database queries and such a query typically requires 5-10 msecs. The in-memory search process for the optimum can be done in a few milliseconds. Even more, we can also exploit 'cross-session' re-use and caching of partial query results, e.g., the set of products that fulfil $usb = true$ remains static as long as no new products appear in the catalog. If no variables are used in the filter rules (like a customer-specified value), relaxations can be computed even without further database accesses.[5]

In our system, the relaxations are also used to 'explain' the proposal, i.e., we use the explanatory text (fragments) for assembling a human-readable explanation and, furthermore, let the user interactively state his actual preferences on the compromises by giving him the possibility to enforce the application/relaxation

[5] To the author's knowledge, no 'benchmark' problems are yet available for comparing these running times.

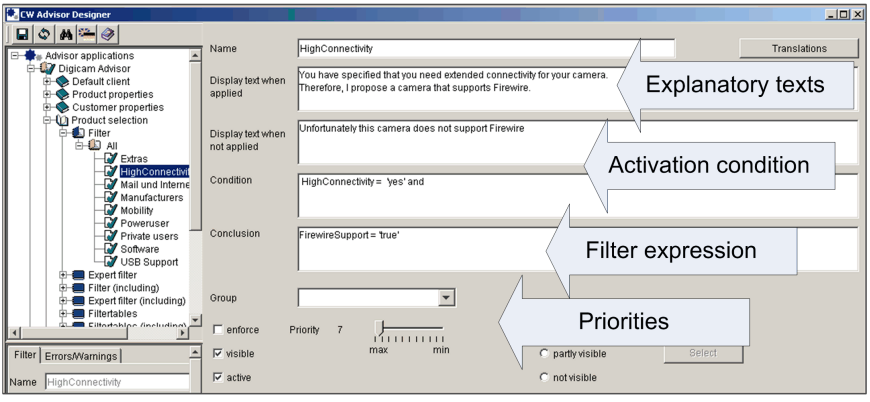

Fig. 10. Graphical editing tool for filter rules

of a rule. An evaluation of a real-world application [4] also indicates that the explanations provided by the system are a well-appreciated feature of advisory applications. Finally, the choice whether an incremental procedure is appropriate has to be decided based on the individual application, e.g., asking additional questions after a longer preference elicitation dialog may for instance be problematic.

5 Conclusion and Future Work

We have presented new techniques and algorithms for fast query relaxation for content-based recommender systems that particularly aim at minimizing the number of required database queries and take a-priory or learned preferences into account. Based on the initial work and formalisms from [9,12] and [3] we have shown how we can a) compute *preferred* conflicts for an interactive relaxation procedure with the help of a recent, general-purpose conflict detection algorithm and b) how the individual evaluation and caching of partial queries allows us to compute optimal relaxations in-memory at the cost of only slightly increased memory requirements. Overall, our approach also continues recent research from the field of Case Based Reasoning (CBR) recommender systems aiming at overcoming the typical shortcomings of such systems [8,13] like for instance the lack of adequate explantation mechanisms.

In our future work, we aim at going one step further than viewing relaxation only as a problem of 'removing' parts of the query: We consider the current approach of fully giving up individual requirements only as a first step in intelligent, content-based recommender systems. In future systems, however, we will therefore aim at building systems that are also capable of coming up with a personalized proposal of how to 'soften' the requirements, e.g., by proposing to increase the price limit by a certain amount.

References

1. S. Amer-Yahia, L. V. S. Lakshmanan and S. Pandit. FleXPath: flexible structure and full-text querying for XML, Proceedings ACM SIGMOD International Conference on Management of Data, Paris, 2004, pp. 83-94.
2. D. Bridge. Product recommendation systems: A new direction. In R. Weber and C. Wangenheim, eds., Workshop Programme at 4^{th} Intl. Conference on Case-Based Reasoning, 2001, pp. 79-86.
3. P. Godfrey. Minimization in Cooperative Response to Failing Database Queries, International Journal of Cooperative Information Systems Vol. 6(2), 1997, pp. 95-149.
4. D. Jannach. ADVISOR SUITE - A knowledge-based sales advisory system. In: Proceedings of ECAI/PAIS 2004, Valencia, pp. 720-724.
5. D. Jannach, J. Liegl. Conflict-directed relaxation of constraints in content-based recommender systems, Proc. 19th International Conference on Industrial, Engineering & Other Applications of Applied Intelligent Systems (IEA/AIE'06), Annecy, France, 2006 (forthcoming).
6. U. Junker. QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. Proceedings AAAI'2004, San Jose, 2004, pp. 167-172.
7. D. Lee. Query Relaxation for XML Model, Ph.D Dissertation, University of California, Los Angeles, June 2002.
8. D. McSherry. Explanation of Retrieval Mismatches in Recommender System Dialogues, ICCBR Workshop on Mixed-Initiative Case-Based Reasoning, Trondheim, 2003, pp. 191-199.
9. D. McSherry. Incremental Relaxation of Unsuccessful Queries, Proc. of the European Conference on Case-based Reasoning, In: P. Funk and P.A. Gonzalez Calero (Eds.) LNAI 3155, Springer, 2004, pp. 331-345.
10. D. McSherry. Maximally Successful Relaxations of Unsuccessful Queries. Proceedings of the 15th Conference on Artificial Intelligence and Cognitive Science, Castlebar, Ireland, 2004, pp. 127-136.
11. R. Reiter. A theory of diagnosis from first principles, Artificial Intelligence, 32(1), 1987, pp. 57-95.
12. F. Ricci, N. Mirzadeh and M. Bansal. Supporting User Query Relaxation in a Recommender System, Proceedings of the 5^{th} International Conference in E-Commerce and Web-Technologies EC-Web, Zaragoza, Spain, 2004.
13. B. Smyth, L. McGinty, J. Reilly, K. McCarthy, Compound Critiques for Conversational Recommender Systems, IEEE/WIC/ACM International Conference on Web Intelligence(WI'04), Maebashi, China, pp. 145-151.