Finding Preferred Query Relaxations in Content-based Recommenders

Dietmar Jannach

Abstract— **In content-based recommender systems, product proposals are generated by exploiting deep knowledge about the items in the catalog. In many implementations of such systems, the users' requirements are directly viewed as constraints that all items in the proposal must fulfill and determining the set of suitable products thus corresponds at least initially to constructing an adequate query to the catalog. In such approaches, however, the problem can easily arise that the catalog query fails because none of the items in the catalog fulfils all of the user's constraints. One general way of dealing with such situations is to relax the catalog query by eliminating individual subqueries and to search for items that fulfill as many constraints as possible. Finding such 'maximal succeeding subqueries' (XSS), however, is not a trivial problem because not all of the potentially many XSSs for a failing query are equally suitable for the user, which means that determining one arbitrary XSS is not sufficient in realistic settings.**

In this paper we present a new technique for determining all maximal succeeding subqueries of a query in an efficient way which allows us to determine optimal or 'preferred' solutions within the limited time frames of interactive recommendation sessions. By evaluating the individual subqueries independently in advance and combining these partial results, we can compute all XSSs in a way that no further costly catalog queries are required. The approach has been implemented in the knowledge-based Advisor Suite recommender system and has been successfully evaluated in several real-world recommender applications.

Index Terms— **Recommender systems, User interfaces**

I. Introduction

Recommender systems are interactive software applications that support the online customer in his/her decision making and buying process. In *content-based* approaches to product recommendation, the product proposals are generated on the basis of detailed descriptions of the items in the catalog: According to Bridge [2], case-based, utility-based, or knowledge-based recommender systems basically fall into this category. Although there may be different techniques involved for eliciting customer requirements, ranking the products, or finding similar items, in many implementations of such systems, some or all of the customer's requirements are - at least initially - viewed as constraints that the items in the proposal have to satisfy [7]. However, when the initial selection of items is based on such a query to the catalog, situations can easily arise, in which none of the products in the catalog fulfills all of the requirements. One basic approach to recover from such situations is to search for items that fulfill as many constraints as possible, which can be achieved by incrementally eliminating one or more constraints from the query (*query relaxation*). In [7], McSherry proposes an incremental, mixed-initiative approach to query relaxation based on results that were achieved in the area 'cooperative query answering' [3]. In his work, he maps the problem of finding items that fulfill as many constraints as possible to the problem of finding a 'maximal succeeding subquery' (XSS) of the original query. In addition, McSherry proposes the computation and utilization of 'minimal failing subqueries' (MFS) and let the user decide in an incremental process, on which part of the query he/she is willing to compromise. Nonetheless, when using the recovery algorithm described in [7], it cannot be guaranteed that the smallest possible or an optimal relaxation with respect to some function describing the 'costs' of the compromises will be found. In order to find an optimal relaxation, in general all XSSs/MFSs of the query have to be known or enumerated, a problem which was shown to be NP-hard in general [3]. In fact, even small-sized problems soon become intractable, in particular if we consider the hard real-time requirements of interactive recommender applications.

In this paper, we propose an algorithm in which the individual subqueries of the original query are evaluated independently in advance and the set of all XSSs can be enumerated by combining these partial results without further costly query operations. The algorithm requires exactly n queries to the catalog for finding all minimal and the optimal relaxation of a query consisting of n subqueries; the additional memory requirements for storing the partial results are also limited. Our approach therefore improves existing work in the area in two directions: First, it reduces the number of database queries for computing possible relaxations to a fixed number which is required in time-bounded interactive recommender applications. In addition, in contrast to previous work in which the size of the relaxation was the main optimization criterion, our technique also supports the concept of *preferred relaxations*.

The paper is organized as follows. After giving an introductory example in the next section, the formal foundations of the approach are summarized. We then describe our algorithm for fast enumeration of all XSSs and afterwards discuss details of the implementation and the evaluation which was done in several real-world recommender applications. The paper ends with a discussion of previous work in the area and an outlook on future extensions.

II. Example

We will illustrate the relaxation problem and our approach with a simplified example from the domain of digital cameras. Our product database consists of the products $p1...p4$ which are characterized by different properties as shown in Figure 1.

D. Jannach is with the Department of Computer Systems, University Klagenfurt, Universitätsstrasse 65, 9020 Klagenfurt, Austria (e-mail: dietmar.jannach@uni-klu.ac.at.)

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. 1. Product database of digital cameras.

Let us assume that the user's requirements can be expressed with the following query to the database, which unfortunately *fails* given the set of available products.

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

Query relaxation will now be viewed as the problem of finding a *maximal succeeding subquery (XSS)*[3] of Q in order to retrieve products that fulfill as many constraints as possible; the difference between the original query and an XSS is called a 'minimal relaxation'. In current approaches to query relaxation the original query Q is split into subqueries according to the attributes which are used in the query ($Q1$ to $Q4$ in our example). The search space in the relaxation problem is determined by the number of these subqueries, i.e., if a query can be divided into n such subqueries, there exist $(2^n - 2)$ candidates that theoretically have to be examined[1]. The lattice of the possible subqueries for our example is illustrated in Figure 2. Testing each of the possible combinations individually is in general not possible in realistic applications because in typical implementations of such systems, each test corresponds to a query to the catalog, i.e., a database query.

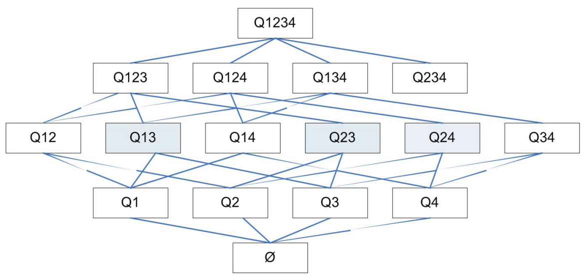
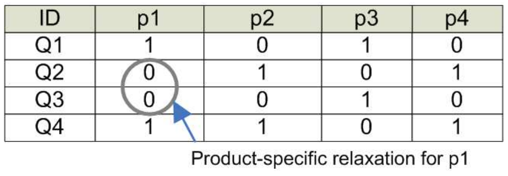


Fig. 2. Lattice of possible subqueries, minmal relaxations are printed in shaded boxes.

In [8], an approach for enumerating all maximal succeeding subqueries is described in which the search space is reduced by a) constructing the subqueries in decreasing order with respect to query length and b) by removing subqueries of already succeeding subqueries from the remaining search space. Still, if we apply this on our example problem, only the single-element subqueries will be pruned from the search space. We therefore propose an approach, in which we can limit the number of needed catalog queries to the number of subqueries of the original query Q, which means that in our case we will only need four queries. We can achieve this by first evaluating the individual subqueries individually and then analyzing and combining the partial results in memory.

[1]We do not have to examine the original query and the empty query.

The results for the subqueries for our example are illustrated in Figure 3: A '1' in the matrix means that the a product will be returned by the subquery; '0' means that the product will be filtered out.



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

Product-specific relaxation for p1

Fig. 3. Evaluating the subqueries individually.

For determining all maximal succeeding subqueries of a query Q we can now proceed as follows. For each product p_i, we can directly infer which of the subqueries causes p_i to be filtered out, which we denote as *Product-specific Relaxation* PSX(Q,p_i), e.g., PSX(Q,p1) = {Q2,Q3}, PSX(Q,p2) = {Q1,Q3} and so forth. In fact, each PSX(Q,p_i) is also a possible relaxation for the overall problem: If we remove all elements of PSX(Q,p_i) from the original query, at least product p_i will satisfy the remaining requirements. Still, not all PSX(Q,p_i) are minimal relaxations, but determining all minimal relaxations can be easily achieved by iterating over all PSX(Q,p_i) once as follows: If the current PSX(Q,p_i) is a superset of an already found relaxation, ignore it. If not, remember it and remove all those relaxations that were already found and for which the current PSX(Q,p_i) is a subset.

Having determined all maximal succeeding subqueries, we can then select the one that promises to be the most useful for the customer. Without having a 'cost model' for the individual parts of the query, a suitable strategy will be to use the relaxation with the smallest cardinality. If such a cost model exists (e.g., information that users rather tend to compromise on the make than on the price), we can also determine the relaxation that minimizes this cost function. Overall, the main difference compared to previous approaches ([7], [8]) is that we do not search in the exponentially growing set of possible combinations of subqueries (the lattice), in which a lot of unnecessary checks are required.

III. Determining maximal succeeding subqueries

We will now describe the relaxation problem and our algorithms more formally. We base our work on the formalisms introduced in [3], [7], and [8].

Definition 1: (Query): A query Q is a conjunctive query formula, i.e., $Q \equiv A_1 \wedge ... \wedge A_k$. Each of the $A's$ 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 conditions $A_1 \wedge ... \wedge A_k$, 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.

Note that in [7] and [8], queries are split into subqueries according to the attributes that the query involves. Within our approach, however, such a specific form of partitioning is not required. Thus, in our approach it also allowed that an individual query atom consists of a disjunction of conditions.

Valid and minimal relaxations and maximal succeeding subqueries are defined as follows.[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.

Maximal succeeding subqueries in the sense of [3] are directly related to minimal relaxations.

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.

The approach described in this paper aims at minimizing the number of queries to the catalog by analyzing the partial results that are obtained by evaluating the subqueries individually.

Definition 6: (Partial query results - PQRS:) Let $P = \{p_1, ..., p_n\}$ be the set of products in the catalog. Given a query Q consisting of atoms A_1, ..., A_k, then $PQRS(A_i, P)$ is a function that describes the subset $P' \subseteq P$ for which condition A_i holds, $i \in \{1, ..., k\}$.[3]

Lemma 3: Given a failing query Q and a non-empty product catalog P, a relaxation R for Q always exists.

For determining the partial query results for a query Q, exactly $|Q|$ queries to the catalog are required. Our algorithm for determining all minimal relaxations works by analyzing the possible relaxations for each product by exploiting the partial query results.

Definition 7: (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.

Determining whether an atom $a_i \in A_1$, ..., A_k of a query Q is part of the product-specific relaxation for p_i can be done without further queries to the catalog by evaluating whether $p_i \in P$ is contained in the partial query result $PQRS(A_i, P)$. If p_i is not contained in $PQRS(A_i, P)$, then A_i has to be part of the product-specific relaxation $PSX(Q, p_i)$.

[2]Note that the term 'relaxation' in the context of this work means *eliminating* parts of the query rather than asking the user to revise his constraints like, e.g., in [1].

[3]$PQRS$ can be represented as a matrix of zeros and ones as shown in Figure 3.

Based on these definitions, we now describe an algorithm for determining all minimal relaxations of a query Q.

```
ALGORITHM: MinRelax
In: A query $Q$, a product catalog $P$
Out: Set of minimal relaxations for Q
PQRS = compute the partial query results for all atoms
    $a_i$ of $Q$ for the product catalog $P$
MinRS = $\emptyset$
forall $p_i \in P$ do
    PSX = Compute the product-specific relaxation
            $PSX(Q, p_i)$ by using PQRS
    % Check relaxations that were already found
    SUB = $\{r \in MinRS \mid r$ is subquery of $PSX\}$
    if $SUB \neq \emptyset$
        % Current relaxation is superset of existing
        continue with next $p_i$
    endif
    SUPER = $\{r \in MinRS \mid$ PSX is subquery of $r\}$
    if $SUPER \neq \emptyset$
        % Remove supersets
        $MinRS = MinRS \setminus SUPER$
    endif
    % Store the new relaxation
    $MinRS = MinRS \cup PSX$
endfor
return MinRS
```

Fig. 4. Algorithm for determining all minimal relaxations

Algorithm $MinRelax$ is sound and complete, i.e., it only returns minimal relaxations and it does not miss any of the minimal relaxations.

Theorem 1: Given a failing query Q and a product database P containing n products, at most n minimal relaxations can exist.

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 than $|PSX| = n$ such minimal relaxations can exist. ■

Proposition 1: 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 Theorem 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. ■

Complexity issues. In our algorithm, the number of required executions of the typically most costly operation - querying the catalog - is equals to the number of atoms of the original query. The other operation that potentially induces relevant computation times is the determination of the subset and superset property when iterating over the products. In the theoretically worst case, at each iteration i this check has to be done for all of the previously found $i-1$ relaxations. This means that if we have n products, $(n*(n+1))/2$ of such checks have to be done in the worst case. However, such checks can be efficiently done in memory and our experiments show that in realistic cases the number of actual checks that have to be made is at least an order of magnitude lower than the theoretical upper bound.

The efficiency of the algorithm with respect to the number of required queries comes at the price of a slightly increased *space complexity* for storing the partial results: For each of the individual atoms of the query, the list of matching products has to be stored. However, there exist p products and the query consists of a atoms, we need *at most* $p*a$ bits for storing the raw information when using, e.g., a representation based on bit-sets.

Searching preferred relaxations. Up to now, we have assumed that relaxations of smaller size, i.e., those who contain fewer conditions to be removed from the query, are preferable for the user. In an interactive recommender application we therefore might pick one of the smallest relaxations and present it to the user. However, as also mentioned in [7], the users might have different preferences on which product characteristics they are more willing to compromise. Such specific preferences can be incorporated into our approach by associating *costs* (of relaxation) with each atom of the query and defining an overall cost function that for instance takes both the number of atoms in the relaxation and these individual costs into account.

Definition 8: (Cost function) Let Q be a failing query and R a valid relaxation for Q consisting of the atoms $a_1, ..., a_k$. If $ICOSTS$ is a function that associates a positive integer number with each a_i expressing the *individual costs* of relaxing atom a_i, the overall costs for R for Q can be described by any function $COSTS(Q, R, ICOSTS)$ that returns a positive integer number expressing the overall costs of R. In addition it has to hold that $COSTS(Q, R', ICOSTS) < COSTS(Q, R, ICOSTS)$ *if* $R' \subset R$ for ensuring that adding additional atoms to a relaxation does not decrease the cost value.

Given such a cost function, we can define an ordering between the possible relaxations and describe the properties of an *optimal* relaxation.

Definition 9: (Optimal relaxation): Given a failing query Q, a valid relaxation R for Q is said to be optimal, if there exists no other valid relaxation R' for which $COSTS(Q, R', ICOSTS) < COSTS(Q, R, ICOSTS)$

Determining the optimal relaxation based on our PSX-representation can be easily done by scanning the set of PSXs, evaluating the cost function individually and remembering the PSX that minimizes this cost function.

Besides searching only for the optimal relaxation, it is also easily possible to determine an ordering among the relaxations for those cases, in which we want to present the user a list of possible relaxations he/she can choose from. Such an ordering can be achieved by sorting the PSX's according to their costs and by removing supersets of already found PSX's.

Note that the cost value for relaxing an individual atom of the query can come from different sources: They can be defined in advance, they could be derived from previous recommendation sessions by taking into account which compromises users typically prefer, or the user could also be directly questioned about his/her personal preferences.

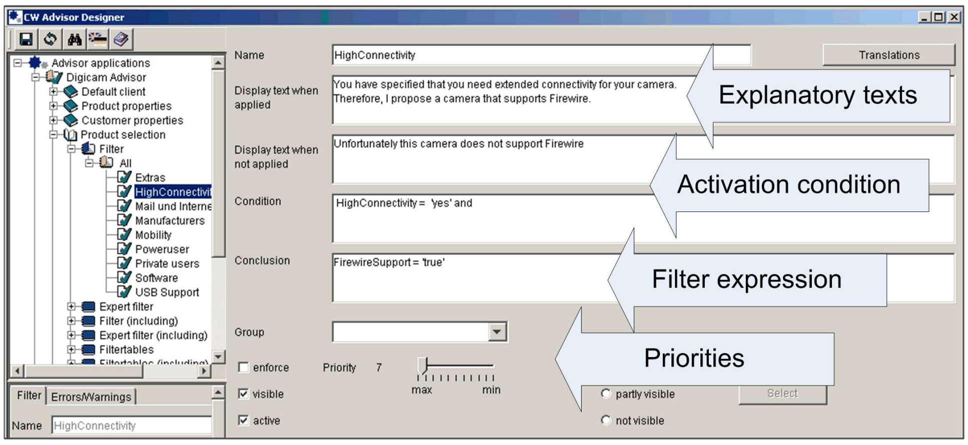
IV. Evaluation

The proposed technique has been implemented and evaluated within the Advisor Suite system (see, e.g., [4], [5]), a fully knowledge-based framework for the development of interactive recommender applications. In this system, the initial set of products to be presented to the user is determined with the help of 'if-then-style' *filter rules* that relate customer requirements with product characteristics. This indirection allows us to implement a more user-oriented interaction view, such that the (non-experienced) users do not have to be questioned directly about desired product characteristics. If we consider the example from Section II, we would, for instance, not ask the user about his need for 'Firewire' support, but rather try to find out what his/her mobility and connectivity requirements are. Typical *filter rules* for our example problem could be the following:

F1: **if** high-quality-printouts are required **then**
only recommend cameras with a resolution > 5MP
F2: **if** user entered price limit L **then**
only recommend cameras that cost less than L
F3: **if** user needs high connectivity **then**
only recommend cameras that support 'Firewire'

At run-time, when a product proposal has to be generated - which is typically done after an initial requirements elicitation phase - the *filter rules* are evaluated by the system: For each rule it is determined, whether the condition in the antecedent of the rule is fulfilled, i.e., whether the filter rule is *active* or not. The conclusions of those active rules are then used to construct a conjunctive query to the catalog. Note that the conclusions of the rules can contain arbitrary complex expressions on product characteristics, e.g., consisting of several conjunctions and disjunctions.

This modular way of modeling recommendation rules also forms the basis for splitting up the conjunctive query into individual atoms for relaxation in a natural way. The filter rules themselves are modeled in Advisor Suite with the help of graphical editing tools (see Figure 5). Each



Name: HighConnectivity

Display text when applied: You have specified that you need extended connectivity for your camera. Therefore, I propose a camera that supports Firewire.

Display text when not applied: Unfortunately this camera does not support Firewire

Condition: HighConnectivity = 'yes' and

Conclusion: FirewireSupport = 'true'

Group:

☐ enforce Priority 7

☑ visible

☑ active

○ partly visible

○ not visible

Name: HighConnectivity

Fig. 5. Graphical editing tool for filter rules

rule can also be annotated with explanatory texts which are presented to the user in the explanation phase. A text can be maintained both for the case when the rule was successfully applied as well as for the case that the rule had to be relaxed. Finally, we can define an a-priori *priority value* for each rule, which corresponds to the costs of relaxing the rule in cases that no product fulfills all of the requirements. In practical applications, these priority values are defined by the domain expert who for instance knows that typical customers rather compromise on the manufacturer of a camera than on other characteristics.

Up to now, about twenty different recommender applications for various domains have been built with ADVISOR SUITE and have been successfully deployed in commercial settings, which gives us in particular a good impression of the size and complexity of realistic knowledge bases:

- The number of products available in the catalog typically ranges from a few dozen to a several hundred.
- The number of filter rules remains manageable, i.e., only a few dozen rules were required in nearly all cases.
- Many of the rules are mutually exclusive with regard to their activation condition, i.e., only a smaller part of the rules is actually *active* when relaxation has to be done.

Running times. The first aspect we have evaluated are the running times for determining (optimal) relaxations: ADVISOR SUITE is a fully Java-based system that operates on top of standard relational database systems; all tests and measurements have been performed on standard desktop PCs with a 'Pentium M 2 GHz' processor and 512 megabytes of RAM.

The most costly operation when determining relaxations in our approach are the queries that are required to compute the result sets for the individual filters, i.e., for each *active* filter, exactly one query has to be executed. The query time mostly depends on the number of products in the catalog and to a smaller extent on the complexity of the query. On average, a single query takes around 5 milliseconds in test cases with around 500 products.

If we assume that we have a knowledge base containing 70 filter rules, and 35 of them are active - which is already a rather hard case in realistic settings - the time for computing the individual filter results is $35 * 5ms = 175ms$.

The results of the individual filter rules are represented in memory in the form of *Java BitSets*, which means that a) the memory requirement for the raw data is limited to *NumberOfProducts* * *NumberOfActiveFilters* bits, and b) that the analysis of the partial results can be efficiently done based on fast bit-set operations.

In all of the recommender applications that are in productive use, we follow a strategy in which we initially compute only one *optimal* relaxation with respect to size and relaxation costs, which is subsequently used to explain the proposal to the user. The time needed for determining this optimum depends on the complexity of the cost function and the number of products. However, these operations can be performed in-memory and in our test cases they required about a tenth of the query time (e.g., 17.5ms in the above-mentioned example).

Regarding query time, we also exploited a particularity of the ADVISOR SUITE approach of modeling *filter rules*: In many cases, the consequent of the rules contains no 'variables' (e.g., *'attribute usb must be true'*). Therefore, we can pre-compute and cache the partial results for such rules in advance, e.g., when the server is started up. In addition, these partial results can also be shared among different recommendation sessions of different users, since the partial results remain stable as long as the filter rule is not changed and the set of products is the same. Such pre-computation, however, is not possible if the consequent contains variables, like in filter rule $F2$ in the example above that takes the user input with respect to allowed costs directly into account. Still, our experiences from different application domains show that the major part of the filter rules do not contain such variables, which means that the number of needed queries can be significantly reduced. Overall, even if no such pre-computation was done, in none of our test cases more than 500 ms were required for finding the optimal relaxation.

Usability aspects. In all our fielded recommender applications, we have adopted a strategy in which we immediately compute a relaxation when we recognize that no product satisfies all constraints that were elicited in the advisory dialog. Thus, the relaxation process is not visible for the user in the first place, since in all cases one or more products will be recommended. Only when the user asks *'Why this recommendation?'*, the proposal is explained with the help of the natural-language text annotations of the filter rules: 'Pro'-arguments correspond to filters that could be applied; 'con'-arguments correspond to the relaxed ones.

This initial relaxation is computed based on the a-priori priorities that were maintained by the domain expert. Since these priorities may not match the customer's actual preferences, the user is then given the possibility to interactively manipulate the priorities: In particular, the user can choose one or more of the relaxed filter rules and state that these rules should not be relaxed. After that, a new relaxation is computed and an alternative proposal is made. Of course, if the user enforces the application of too many filter rules, an empty result can be the consequence. In such situations, the user can *undo* his/her decisions and evaluate other alternatives.

For one of the applications built with ADVISOR SUITE a detailed study was performed [9] in order to evaluate the usability of the overall system. The study of an application hosted on Austria's largest e-Commerce site (with respect to daily visits) included the analysis of more than 100.000 recommendation sessions and about 1.600 feedback forms. Although this study was not primarily concerned with the relaxation facilities, we could learn in particular from the feedback forms that the system's capability to produce explanations was the feature that was appreciated most by the users [4]. Even more, from the interaction logs we could see that many visitors made use of the possibility to evaluate different alternatives of relaxations, in cases when their original requirements could not be fulfilled.

Note that in the fielded applications we did not allow the users to *fine tune* the priorities by themselves, e.g., by assigning a value between 1 and 10 for each rule, because an in-house assessment showed that such a task may be too complex for most users and it is also complicated to explain the effects of changing these priorities to the user. However, our future work includes the incorporation of *self-adapting* priorities, i.e., a mechanism that tracks the users' behavior over a given time frame and learn the typical user preferences.

V. Summary

In this paper we have shown how optimal relaxations for unsuccessful queries in the context of interactive, content-based recommenders can be efficiently computed by pre-evaluating and analyzing the individual subqueries of the failing query. The proposed approach has been implemented in a domain-independent framework for building knowledge-based recommender systems and was evaluated with the help of several real-world recommender applications. Our measurements showed that even hard test cases can be successfully solved within the tight time frames that we have to deal with in interactive recommendation sessions. An empirical evaluation suggests that relaxation and explanation features are well appreciated by the online users as long as no complex interaction sequences are required.

Our work is based on the formalisms and relaxation approach also used by Godfrey [3] in the context of 'Cooperative Query Answering'. The goal in that research field is to establish a basis for building database systems that are capable of returning *more informative* answers than only 'yes' or 'no' to the users' queries. In principle, the algorithms presented in [3] could also be applied for our specific purposes in the recommendation domain. Nonetheless, the search algorithms for XSSs from [3] do not exploit partial pre-computations for adequate run-time behavior; furthermore, no ranking of relaxations based on preferences is possible, i.e., the ranking is restricted to cardinality only.

The work from [7] and [8] is similar to ours with respect to the overall goal; the approaches presented in this paper can be seen as algorithmic improvements that take specific characteristics of the domain into account, e.g., the limited number of records in the catalog, for guaranteeing short response times. In addition, *user preferences* can be directly taken into account for optimization purposes. Compared with [7] and [8], we also claim that our approach is more flexible with regard to how the query can be split into subqueries in a natural way, which is done in [7] on the basis of query attributes only.

Our future work will include further research towards 'self-adapting' systems, where priorities and preferred relaxations can be *learned* from different sources of knowledge like past user behavior. An recent extension of our approach, which allows us to determin optimal relaxations that comprise *at-least-n* products, can be found in [6]; in this work, also a new algorithm for fast interactive, user-driven relaxation (comparable to [7]) is proposed.

References

[1] D. Bridge. Towards Conversational Recommender Systems: a Dialogue Grammar Approach. In: D.W. Aha (ed.) Proceedings EWCBR-02 Workshop on Mixed Initiative CBR, pp. 9-22, 2002.

[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, Spain, IOS Press, 2004, pp. 720-724.

[5] D. Jannach, G. Kreutler. A Knowledge-Based Framework for the Rapid Development of Conversational Recommenders. In: X. Zhou, S. Su, M. Papazoglou, M. Orlowska, K. Jeffery (Eds.): Web Information Systems - WISE 2004. Springer LNCS 3306, 2004, pp. 390-402.

[6] D. Jannach. Techniques for Fast Query Relaxation in Content-based Recommender Systems, Proceedings of Annual German Conference on AI 2006, Bremen, Springer LNCS, to appear, 2006.

[7] D. McSherry. Incremental Relaxation of Unsuccessful Queries, Proceedings of the European Conference on Case-based Reasoning, In: P. Funk and P.A. Gonzalez Calero (Eds.) Lecture Notes in Artificial Intelligence 3155, Springer, 2004, pp. 331-345.

[8] 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.

[9] M. Zanker, C. Russ. Geizhals.at: vom Preisvergleich zur E-Commerce Serviceplattform, in S.M. Salmen, M. Gröschel: Handbuch Electronic Customer Care, Physica, Heidelberg, 2004.