Answering Queries based on Preference Hierarchies

Georgia Koutrika
Stanford University, USA
koutrika@stanford.edu

Yannis Ioannidis
University of Athens, Greece
yannis@di.uoa.gr

ABSTRACT

People's preferences are expressed at varying levels of granularity and detail as a result of partial or imperfect knowledge. One may have some preference for a general class of entities, e.g., liking comedies, and another one for a fine-grained, specific class, e.g., disliking recent thrillers with Al Pacino that are intended for families. In this paper, we are interested in capturing such hierarchical preferences to personalize database queries and in studying their impact on query results. In particular, we organize the collection of one's multi-granular preferences in a *preference hierarchy* (a directed acyclic graph), where each node refers to a subclass of the entities that its parent refers to, and whenever they both apply, more specific preferences override more generic ones. We study query personalization based on preference hierarchies and provide efficient algorithms for identifying relevant preferences, modifying queries accordingly, and processing these queries to obtain personalized answers. Finally, we present results of experiments both synthetic and with real users, which (a) demonstrate the efficiency of our algorithms, (b) provide insight as to the appropriateness of the proposed preference model and (c) show the benefits of query personalization based on hierarchical preferences compared to flat preference representations.

1. INTRODUCTION

In an ideal world, preferences could exist for every object in a domain of interest, e.g., movies, books, etc. Such elaborate preferences would yield a perfectly fine-grained ranking of alternatives and highly personalized content. In another ideal world, one could express preferences for general but well-defined, disjoint sets partitioning the objects of discourse. Unfortunately, neither of these two extremes is found in practice often. User preferences are typically incomplete and our knowledge of them is imperfect and partial. General and specific preferences coexist peacefully.

People tend to have a mix of general and specific preferences. This mix may indicate lack of knowledge, lack of elaborate taste, or inability to identify those properties of objects that determine their preferences. For instance, one may have a general liking for adventures but a finer-grained taste only for a subset of them, such as those directed by S. Spielberg, which are characterized as favorites. On the other hand, one may have liked some particular books but may be unable to identify a common characteristic that made them attractive. Implicit collection of user preferences, for instance by observing user behavior in the system, suffers from similar problems. For instance, consider a system that builds user profiles from user ratings for movies to provide personalized content. Suppose that a user has given high ratings to all comedies she has seen so far. This knowledge allows the system to conclude that the user likes comedies but does not suffice to differentiate among comedies. Suppose that the user continues to interact with the system and provides some low scores for comedies starring Jack Nicholson. Then, the system may enrich its knowledge of this user's preferences by discriminating against comedies starring Jack Nicholson, which will be ranked lower than other comedies. Note that not all forms of additional information are useful. For instance, another user may have liked all comedies with Adam Sandler so far; this does not necessarily imply, however, that the user should like all comedies or all movies with Adam Sandler independently.

In this paper, we are interested in *capturing user preferences at different levels of detail and using them for personalizing the results of database queries.* In particular, we propose a framework that allows a profile to contain any mix of general and specific preferences, where the latter are explicitly stated rather than being implicitly calculated from the former as in, e.g., [2, 17]. We organize the collection of one's multi-granular preferences in a *preference hierarchy* (a directed acyclic graph), where each node refers to a subclass of the entities that its parent refers to, and whenever they both apply, more specific preferences override more generic ones, whether the former represent a stronger or a weaker preference than the latter. Given the increased expressive power and freedom offered by this framework, we study query personalization [17, 20] based on preference hierarchies and provide efficient algorithms for identifying relevant preferences, modifying queries accordingly, and processing these queries to obtain personalized answers. We evaluate the impact of these finer-grained answers on users. Furthermore, we raise the question of whether or not increased expressivity is achieved at the expense of performance and answer it in the negative through an appropriate experimental performance evaluation. In summary, our contributions are:

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '08, August 24-30, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

- We introduce a framework that is based on the concept of preference hierarchies, which allows the representation of generic and specific preferences and their relationships in a lean and flexible way (Sect. 3).
- We present a system architecture and algorithms for query personalization based on preference hierarchies. There are three basic issues: (a) finding preference relationships, (b) constructing a hierarchy of related preferences for a query, and (c) using this hierarchy to generate personalized answers that respect the semantics of query personalization and the relationships between the preferences of the hierarchy (Sect. 4).
- We study the overall impact of the increased expressivity allowed by our framework through several experiments both synthetic and with real users that evaluate the preference model and the algorithms. We show that more accurate, finer-grained result rankings can be achieved without losing in performance (Sect. 5).

2. RELATED WORK

Preferences have received attention from researchers in different disciplines (e.g., [10, 6]). In IR and Databases, preferences are associated with the search process either as explicit preferences entered through a query interface, long-term preferences gained with preference mining [11] or preferences based on user feedback [4]. IR-based representations of preferences include bags of words [13], vectors of terms [3] etc. In databases, early efforts go back to eighties [18]. Since then many approaches followed falling into two categories. In *qualitative* approaches, preference relations are defined using logical formulas [8] or special preference constructors [15] and are embedded into relational query languages through a relational operator that selects from its input the set of the most preferred tuples (winnow [8], BMO [15]). Several algorithms for computing skyline queries, which are special cases of preference queries, exist (e.g., [5]). *Quantitative* approaches aim at an absolute formulation of preferences. Preferences in queries are specified indirectly using scoring functions that associate a numeric score with every tuple of the query answer [2]. Several algorithms have been proposed for the efficient computation of top-K objects (e.g., [21]). Preferences that hold only in specific contexts have been also studied [12, 20].

Our work is closer to [17, 20] in that we share the idea of representing preferences as query conditions associated with a degree of interest for personalizing queries. In contrast to these efforts that focus on other problems of preference representation, such as differentiating preferences based on their intensity [17] and studying context-dependencies [20], we focus on the *complexity* of preferences and its implications. One critical departure from these works is that we lift the central assumption that preferences hold independently of each other. We capture preferences of different granularity and we define preference relationships on the basis of preference mappings. The latter are essentially based on the concept of containment mappings for conjunctive queries [1, 7] but they are simpler because they can be resolved with 1 to 1 mappings of atomic conditions in the preferences compared. Finally, we experimentally compare our algorithms with their simpler counterparts, which assume preference independence.

3. FRAMEWORK

We consider that for every user there is a profile storing user preferences for personalizing queries. Without loss of generality, we focus on `SPJ` queries over relational databases.

3.1 Preference Formulation

We consider a database $\mathcal{D}$. R and A are used to denote a relation and an attribute, respectively, in the database. We use the symbols R_i on some index i and A_j on some index j, when more than one relation or attribute are discussed. A set of attributes is denoted $\mathcal{A}$.

A *preference* for a set of tuples of a relation R in the database $\mathcal{D}$ is expressed as a degree of interest $d \in [0, 1]$ in a condition q that describes the qualifying tuples in R, and is denoted $R : (q, d)$. $d = 0$ indicates no interest, and $d = 1$ indicates extreme interest. Depending on the form of condition q, we distinguish the followin types of preferences:

`Atomic preference` If q is a single, atomic, selection or join, condition, then a preference for q is called atomic.

`Composite preference` If q is a conjunction of multiple atomic conditions, then a preference for q is called composite.

`Selection preference` Consider a set $\mathcal{A}$ of attributes transitively connected to R based on joins on the database graph. For any conjunction q of atomic selections involving the attributes in $\mathcal{A}$ and atomic joins transitively connecting these attributes to R, a user's preference for tuples in R satisfying q is expressed by a degree d of interest in q, and is a selection preference.

`Join preference` Consider a conjunction of atomic join conditions q representing the transitive join of relations R_i and R_j. A user's preference for tuples from R_i that join to tuples in R_j is expressed by the degree d of interest in q, and is called a join preference.

Selection preferences indicate interest in attribute values. Join preferences indicate to what degree related entities are mutually influenced by preferences. From a different perspective, that of using preferences to personalize query results, a preference $R : (q, d)$ can be interpreted as follows: given a query involving R, d shows the user interest in considering q for personalizing the query results. Since a preference for a set of tuples is essentially captured as a preference for a condition that describes this set, in what follows, we consider the terms "preference for tuples" and "preference for a condition" equivalent and we use them interchangeably. We use the symbol p or p_i (if a set of preferences is discussed) to refer to a preference.

Preference tree. A preference $R : (q, d)$ can be represented as a *preference tree* $g(V, E)$. This is a rooted tree, which can be thought as an extension of the traditional schema graph, with the following characteristics. Nodes in V map to relations, attributes, and values in q, and can be possibly replicated if the corresponding relation, attribute or value is used more than once in q. Edges in E connect (a) a relation node to another relation node, representing a join condition in q or (b) an attribute to its container relation or (c) an attribute to a value, representing a selection in q. Join edges are tagged with the corresponding joining attributes, and selection edges are tagged with the operator used in the selection condition. R maps to the tree's root. For selection preferences, the tree leaves are always values.

A preference tree can take either of two forms. It can contain one single path mapping to a join preference or it

Table 1: Example Preferences

	User preference	Preference formulation
p_1	*I quite like sci-fi movies directed by S. Spielberg*	MOVIE : MOVIE.mid=GENRE.mid and GENRE.genre='scifi' and MOVIE.mid=DIRECTED.mid and DIRECTED.did=DIRECTOR.did and DIRECTOR.name= 'S. Spielberg' 0.7
p_2	*I love adventure movies intended to families*	MOVIE : MOVIE.mid=GENRE1.mid and GENRE1.genre='family' and MOVIE.mid=GENRE2.mid and GENRE2.genre='adventure' 0.9
p_3	*It is really important to me who is the director of a movie*	MOVIE : MOVIE.mid=DIRECTED.mid and DIRECTED.did=DIRECTOR.did 1.0
p_4	*Hitchcock is a great director*	DIRECTOR : DIRECTOR.name = 'A. Hitchcock' 0.9

can contain a set of paths connecting its root to a set of values capturing a selection preference.

Example. Consider the following toy database.

MOVIE(mid, title, year, duration) GENRE(mid, genre)
CAST(mid, aid) ACTOR(aid, name)
DIRECTED(mid, did) DIRECTOR(did, name, nationality)

Table 1 shows some preferences in natural language (in the left column) and how they can be represented using the preference model (in the right column). Observe how there may be more than one instance of any relation or attribute in a preference, e.g., the relation GENRE and its attribute genre are referenced more than once in p_2. Figure 1(a) depicts the preference tree for p_2. For the sake of presentation, we do not show tags on the edges, since for our toy database they are understood. Observe how repeated relations and attributes are mapped to different nodes in the preference tree. As a notational convention, different occurrences of a relation (mapping to different nodes in the tree) are written as a concatenation of the relation name and a sequential number. To distinguish when an attribute is used with a different occurrence of the same relation, the attribute takes the relation's sequential number in their representation. Figure 1(b) shows the preference tree for p_3, which comprises a single path in contrast to the tree for p_2, which maps to a set of paths from the root to the value nodes. Preference trees can "unfold" complex conditions, such as self-joins, and map them to distinctive paths on the tree. To illustrate, consider the following, rather non-typical, preference for 2008 movies by French directors who have also directed movies released in 2000, whose tree is depicted in Figure 1(c).

MOVIE1.year=2008 and MOVIE1.mid=DIRECTED1.mid and
DIRECTED1.did=DIRECTOR1.did and
DIRECTOR1.nationality='French' and
DIRECTOR1.did=DIRECTED2.did and
DIRECTED2.mid=MOVIE2.mid and MOVIE2.year=2000

Preference Inference. On the basis of the preferences stored in a user profile, one can compose implicit preferences, i.e., preferences that are not explicitly stored in the profile but can be implied by those actually stored. In particular, consider a join preference $R_i : (q_i, d_i)$ connecting R_i to relation R_j, and a join or selection preference $R_s : (q_s, d_s)$. If $R_j = R_s$, then these preferences are *composeable*.

Given a set of composeable preferences, $R_1 : (q_1, d_1)$, $R_2 : (q_2, d_2)$, ... $R_m : (q_m, d_m)$, we compose the *implicit preference* $R : (q, d)$, such that: (a) R=R_1, (b) q is the conjunction of the conditions $q_1, q_2, ...q_m$ and (c) its degree d of interest is a function of the degrees of interest of the base preferences, i.e., $d = f(d_1, d_2, ...d_m)$. d should be a non-increasing function (e.g., the product) of the base degrees of interest, $d_1, d_2, ...d_m$, on the grounds that it cannot

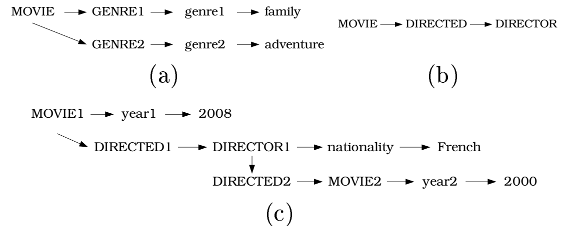


Figure 1: Example preference trees.

exceed the degrees of interest of its supporting preferences. For example, from p_3 and p_4 shown in Table 1, we can imply a preference for movies directed by Hitchcock using the product of the degrees of interest, represented as:

MOVIE : MOVIE.mid = DIRECTED.mid and
DIRECTED.did = DIRECTOR.did and
DIRECTOR.name = 'A. Hitchcock' 0.9

3.2 Preference Hierarchies

Depending on the form of the condition q, selection preferences can be defined on a relation R at varying levels of granularity, ranging from quite generic (e.g., an atomic selection on R) to very specific preferences that combine multiple (atomic or implicit) selections. Intuitively, one may view a selection preference as defining a *class* of entities with some particular features. Given the preferences $R : (q_i, d_i)$ and $R : (q_j, d_j)$, q_i and q_j may define two different classes of entities from R. For example, "MOVIE.year = 2000" and "MOVIE.mid = GENRE.mid and GENRE.genre='comedy'" describe two independent classes of movies: the class of movies released in 2000 and the class of comedies. A movie can belong to both classes, hence, these preferences can *independently hold*. However, the condition "MOVIE.year = 2000 and MOVIE.mid = GENRE.mid and GENRE.genre='comedy'" explicitly refers to a *subclass* of the entities that "MOVIE.year = 2000" refers to. Consequently, whenever they both apply, the more specific one, i.e., the one defining the subclass, *overrides* the more generic one. In what follows, we formally define preference overriding and independence.

We first start with some observations. We can view a preference $R_i : (q_i, d_i)$ as a possible conjunctive query, which selects tuples from R_i that satisfy q_i. On the basis of this correspondence, we can define preference overriding through conjunctive query containment and containment mappings: Given two preferences, $R_i : (q_i, d_i)$ and $R_j : (q_j, d_j)$, we consider the corresponding conjunctive queries Q_i and Q_j. We say that p_i *is overridden by* p_j if Q_i is contained in Q_j ($Q_i \subseteq Q_j$). Q_i is contained in Q_j iff there is a containment mapping from Q_i to Q_j [1, 7]. However preferences map to rooted trees, therefore, it is easily provable that preference containment mappings can be performed in a more

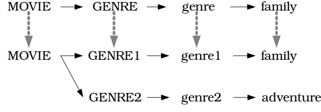


Figure 2: Preference mapping.

straightforward way, i.e., as "1 to 1" mappings of the atomic conditions in the two preferences (Fig. 2 shows an example). On the basis of the above, the definitions of preference overriding and independence follow.

Preference Relationships. Given a set $\mathcal{P}$ of selection preferences and two preferences $R_i : (q_i, d_i)$ and $R_j : (q_j, d_j)$ (denoted p_i and p_j, resp.) from $\mathcal{P}$, we define the following:

`Preference Overriding` If p_i can be mapped to p_j, such that (a) $R_i \equiv R_j$ and (b) each atomic condition in q_i is mapped to an atomic condition in q_j with the same relations and attributes, then p_i *is overridden by* p_j (p_j *overrides* p_i), denoted $p_i \sqsubset p_j$. In this relationship, p_j is *specific*, p_i is *generic*, and the following degree of interest correspondences hold (denoted by $\leftarrow$):

$$doi(p_i \wedge p_j) \leftarrow doi(p_j) \quad (1)$$

$$doi(p_i \wedge \neg p_j) \leftarrow doi(p_i) \quad (2)$$

i.e., the generic preference (p_i) holds only in the absence of the specific one (p_j). If $\nexists p \in \mathcal{P}$ s.t. $p_i \sqsubset p \sqsubset p_j$, i.e., p_j is the most generic specialization of p_i in $\mathcal{P}$, then p_i is *properly overridden* by p_j, denoted $p_i \sqsubseteq p_j$.

`Preference Independence` If $R_i \equiv R_j$ but $p_i \not\sqsubset p_j$ and $p_j \not\sqsubset p_i$, then p_i and p_j are *independent* and it holds that:

$$doi(p_i \wedge p_j) \leftarrow r(doi(p_i), doi(p_j)) \quad (3)$$

i.e., both preferences can hold together.

Given a set $\mathcal{P}$ of selection preferences over the same relation R in the database $\mathcal{D}$, $\mathcal{P}$ is mapped to a hierarchy that reflects the relationships among its preferences, such that each node refers to a subclass of the entities that its parent refers to, and whenever they both apply, the more specific preference overrides the more generic one.

Preference Hierarchy. A *preference hierarchy* $\mathsf{G}_H(\mathcal{V}_H, \mathcal{E}_H)$ corresponding to a set of preferences $\mathcal{P}$ is a directed acyclic graph. Nodes in $\mathcal{V}_H$ map to preferences in $\mathcal{P}$. Given two nodes v_i and v_j in $\mathcal{V}_H$, mapping to preferences p_i, p_j $\in \mathcal{P}$, respectively, an edge in $\mathcal{E}_H$ from v_i to v_j, $e(v_i, v_j)$, denotes that p_i is properly overridden by p_j. When a preference is not overridden by any other preference, it is called a *leaf*. When it does not override any preferences, it is called a *root*. Note that the term "root" is loosely used here; by definition, a hierarchy has one root, while our definition allows more than one root in a preference hierarchy.

Figure 3 illustrates a preference hierarchy (on the left side) corresponding to a set of preferences (on the right side). For clarity, relations are abbreviated, e.g., M stands for MOVIE, DD stands for DIRECTED, and so forth. Roots are the preferences p_1 and p_6 while p_4 and p_5 are leaves. We also observe that the degrees of interest at one level of the hierarchy are not derived from the degrees of interest at upper or lower levels of the hierarchy. For instance, p_2 is a strong preference, whereas the more specific p_5, which differentiates a subset of tuples satisfying p_2, states a weak preference.

3.3 Combining Preferences

In the absence of an explicit preference for a particular tuple in the database, we can compute a degree of interest in this tuple by combining the known preferences for it.

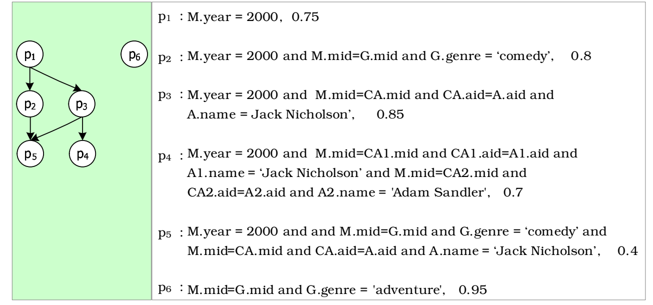


Figure 3: A set of preferences as a hierarchy.

For instance, if we have a preference for movies by S. Spielberg and a preference for adventures, we can compute the combined preference for adventures by S. Spielberg.

Ranking Function. Consider a set $\mathcal{P}'= \{p_i | p_i, i = 1 \ldots n'\}$ of independent preferences over the same relation. The degree of interest in their conjunction is a function r, called *ranking function*, of their degrees of interest:

$$doi(\mathcal{P}') = doi(p_1 \ \wedge \ \ldots \ p_{n'}) = r(doi(p_1), \ \ldots \ doi(p_{n'})) \quad (4)$$

There are many possible ranking functions that could be used, e.g., the maximum or the average, depending on the application, the objects of interest, and so forth. For instance, for choosing movies, ranking them based on the best preference they meet may suffice, whereas when buying a car, where many of the car features matter, a function such as the average of preferences may be more appropriate. However, a tuple may satisfy a set of preferences that are not independent and, hence, cannot be freely combined.

Tuple ranking. Consider a set $\mathcal{P}$ of preferences for the same relation R. The degree of interest in a tuple $t \in R$ satisfying $\mathcal{P}$ is: $doi(t) = doi(\mathcal{P}')$, with $\mathcal{P}' \subseteq \mathcal{P}$ s.t.:

- $\forall p_i, p_j \in \mathcal{P}'$, p_i, p_j are independent and
- $\forall p_i \in \mathcal{P}'$, $\nexists\ p_j \in \mathcal{P}$ that is more specific from p_i.

Example. Consider the preferences shown in Figure 3 and a 2000 comedy with J. Nicholson, which, in principle, satisfies preferences p_1, p_2, p_3 and p_5. Taking into account their relationships, not all preferences count and the degree of interest in this movie m is computed as follows (using the maximum of degrees as the ranking function:)

$$doi(m) = doi(p_1 \wedge p_2 \wedge p_3 \wedge p_5) \overset{p_1 \sqsubseteq p_2}{=} doi((p_1 \wedge p_2) \wedge p_3 \wedge p_5)$$
$$\overset{(1)}{=} doi(p_2 \wedge p_3 \wedge p_5) \overset{p_2 \sqsubseteq p_5}{=} doi((p_2 \wedge p_5) \wedge p_3)$$
$$\overset{(1)}{=} doi(p_5 \wedge p_3) \overset{p_3 \sqsubseteq p_5}{=} doi(p_5) = 0.4$$

4. ALGORITHMS

The query personalization system consists of a number of modules (Figure 4). *Preference Construction* extracts and composes preferences from the user profile that are related to a query. A user can ask for or the system may automatically determine the number of related preferences that should be built from the user profile on the basis of many factors, such as the extent of personalization desired (few vs. many preferences), the number of results desired (satisfying more preferences may generate very focused or even empty results), and so forth. Since there may be generic and specific preferences, the relationships between them need to be established. *Preference-Hierarchy Integration* is responsible for constructing the hierarchy of related preferences for the query. This module places each preference found from the first module in the right "position" in the hierarchy. It collaborates with the *Relationship Finder*, which determines

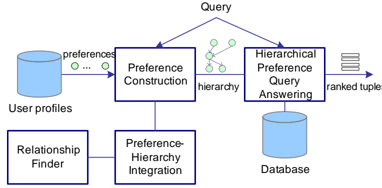


Figure 4: System Architecture.

the kind of relationship for a pair of preferences. Finally, *Hierarchical Preference Query Answering*, takes into account the hierarchy of related preferences and returns results that meet the initial query and the user preferences. Again, the number of preferences that a personalized answer should satisfy can be determined by the user or the system depending on the size of the results returned and the personalized answer's focus desired. For instance, if the initial query is very general and returns many results, then a user may wish to apply a larger number of preferences on the query in order to shape an answer of manageable size.

4.1 Relationship Finder

Given two selection preferences, p and p', defined over the same relation R, there are three mutually exclusive cases: (a) p is overridden by p'; (b) p' is overridden by p; or (c) p and p' are independent. We have seen that preference relationships are defined based on preference mappings (Sect. 3.2). Since preferences map to rooted trees, where the root is a relation and the leaves are always values, the preference mapping problem can be formulated as a *tree matching problem*. While typically tree/graph searching and matching problems are defined as mappings of nodes and edges in two different graphs [9, 19] and are often approximated through matching sets of paths on the graphs compared, the preference tree matching problem is in fact defined as a *one-to-one mapping of the paths that connect the root to the leaves in two preference trees*. Moreover, it does not present the particularities of a general graph search problem since it is contained in the comparison of just two trees. The above observations allow building a matching solution that can correctly match two preference trees by counting the number of matching root-to-leaf paths between the two trees. It is inspired by ATreeGrep, which however counts the number of mismatching root-to-leaf paths between a query tree and a data tree [19]. An additional twist of the preference tree matching is that it needs to find the exact relationship of two trees rather than just tell whether they match or not.

Our approach is the following. For the preferences p and p', we consider the sets *P* and *P'* of all root-to-leaf paths on their preference trees, respectively. $|P|$ and $|P'|$ are the sizes of these sets. For efficient path counting and comparison, we adopt a string representation of a path (string encoding of trees in general has been proposed in [22].) To generate a path representation, we concatenate the names of nodes in the path. For instance, the string representation of p_1 in Figure 3, which comprises a single path, is "Myear2000". Two paths $s_i \in P$ and $s'_i \in P'$ *match* iff their string representations are the same. Then, the relationship of p and p' can be determined by counting the number M of pairs (s_i, s'_i) of matching paths. Setting aside the case that these sets are actually the same, the following hold. If $M = |P|$ it is $p \sqsubset p'$. If $M = |P'|$ then $p' \sqsubset p$. If none of the above holds, then p' and p are independent. This process is captured in the algorithm FIND_RELATIONSHIP. In case of inequalities, the

```
Algorithm FIND_RELATIONSHIP
Input:   preference p, preference p'
Output:  relationship
Begin
1. read the sets P and P' of root-to-leaf paths for p, p', resp.
2. If  |P| ≤ |P'|       {relationship := MATCH(P, P') }
   Else                 {relationship := MATCH(P', P) }
3. output <p relationship p'>
End

procedure MATCH
Input:   set P1, set P2
Output:  relationship
Begin
1.    M := 0; found := true
2.    While found=true and ∃ unexamined path s_i ∈ P1 {
2.1     If s_i matches some s'_i ∈ P2
2.1.1     {M := M+1    }
2.2     Else
2.2.1     {found := false  }                                   }
3.    If M= |P1|
3.1     If M= |P2|
3.1.1     {relationship := 'is the same to' }
3.2     Else
3.2.1     {relationship := 'is overridden by' }
4.    Else
3.1     {relationship := 'is independent from' }
4.    output relationship
End
```

process is slightly different: it matches paths without considering the selection values, and performs an additional check for the atomic selections to determine their relationship. In the presentation of subsequent algorithms, we assume selections with equalities. This assumption does not affect the algorithms in any way but it helps the presentation.

4.2 Preference Construction

In personalizing a query, only selection preferences related to (and not in conflict with) the query are valid for modifying its results. We assume that we are interested in the top K preferences in the order of degree of interest. We define preference relatedness and conflicts on the basis of the syntactic characteristics of the query and the preferences.

Related Preference. A preference $R : (q, d)$ is related to a query Q, if Q already involves R.

Conflicting Preference. A preference $R : (q, d)$ is conflicting with a query Q, if: (a) it is related to the query; (b) when q is inserted into the original query qualification, replacing any part of the latter that coincides with it, i.e., joins or selections on a common attribute, the resulting query qualification and the original one contain at least one common implicit selection; and (c) this selection is specified over a single-value attribute of the query results.

To illustrate, consider a query for fantasy movies by Robert Zemeckis and the preferences listed in Table 1. p_1 is related to but conflicting with the query (assuming that each movie has one director). p_2 is related and not conflicting because GENRE.genre is a multi-value attribute for movies.

Algorithm Hierarchical-RP. Preferences stored in a profile can help us derive implicit preferences. The algorithm HIERARCHICAL-RP does not only return preferences related to a query that are explicitly stated in the profile, but it can also compose related preferences starting from the stored ones and iteratively considering additional stored preferences that are composeable with the ones already known. The algorithm considers a profile U, a query Q, and a limit K on the preferences to be extracted from U, and generates

Algorithm HIERARCHICAL-RP

Input: query Q, a user profile U, K
Output: a preference hierarchy $\mathsf{G}_H(\mathcal{V}_H,\,\mathcal{E}_H)$
Begin
1. $\mathcal{V}_H{:=}\varnothing$, $\mathcal{E}_H{:=}\varnothing$
2. $L := \{(p_i, \mathsf{P}_i) | p_i \in U$ related to Q $\}$
3. **While** $L{<>}\varnothing$ and $K > 0$ {
3.1. remove head (p,P) from L
3.2. **If** p is a selection preference {
3.2.1. add p to $\mathcal{V}_H$; $K = K - 1$
3.2.2. G_H:=HIERARCHICAL_TRAV(G_H, (p,P)) }
3.3. **Else** {
3.3.1. **Foreach** $p_j \in U$ composeable with p {
If $(p \wedge p_j)$ is not conflicting with Q {
$\mathsf{P}'_j{=}\varnothing$
Foreach $s_i \in \mathsf{P}_j$ and $s \in \mathsf{P}$
add $s_i \& s$ in P'_j
add $(p \wedge p_j, \mathsf{P}'_j)$ to L } } }
4. output $\mathsf{G}_H(\mathcal{V}_H,\,\mathcal{E}_H)$
End

a hierarchy of the top K related preferences for Q. For this purpose, it maintains a list L of selection and join preferences that are related to the query ordered in decreasing degree of interest. Initially, this list contains all preferences stored in the profile that are related to the query. At each round, the most significant preference is processed based on its type. If it is a join preference (ln: 3.3), the algorithm considers all stored preferences that are composeable with it to infer new preferences that are related to the query. These are inserted into L. Recall that the degree of interest in an inferred preference is a non-increasing function of the degrees of interest of its supporting preferences (Sect. 3.1). Therefore, all new preferences will not be more significant than the join preference processed. This property ensures that selection preferences are processed in the right order of degree of interest (ln: 3.2), and thus the final output of the algorithm will comprise the top K selection preferences related to the query. Each selection preference found is placed in a hierarchy with the help of HIERARCHICAL_TRAV, which performs the Preference-Hierarchy Integration (Sect. 4.3).

In addition, for each preference, the algorithm constructs the set of root-to-leaf paths from its preference tree. Each path is encoded as a string for the purposes of finding relationships between pairs of preferences, as we have discussed in Sect. 4.1. Building the set of paths for a preference that is stored in the profile is straightforward. When the algorithm builds a new preference by composing a join preference p with another preference p_j (ln: 3.3.1), then the set of paths for the new preference is generated by concatenating each of the paths that map to p_j with the path that maps to p (since p is a join preference, it maps to a single path.)

4.3 Preference-Hierarchy Integration

We study the integration of a preference p into a preference hierarchy $\mathsf{G}_H(\mathcal{V}_H,\,\mathcal{E}_H)$, given that p is defined on the same relation as the preferences in the hierarchy.

Problem statement. Given a preference p and a preference hierarchy $\mathsf{G}_H(\mathcal{V}_H,\,\mathcal{E}_H)$, the result of their integration is a new hierarchy $\mathsf{G}'_H(\mathcal{V}'_H,\,\mathcal{E}'_H)$, such that:

(a) $\mathcal{V}'_H = \mathcal{V}_H \cup \{p\}$,
(b) $\forall p_i \in \mathcal{V}_H$ with $p_i \sqsubseteq p, \exists e(p_i, p) \in \mathcal{E}'_H$, and
(c) $\forall p_i \in \mathcal{V}_H$ with $p \sqsubseteq p_i, \exists e(p, p_i) \in \mathcal{E}'_H$

Algorithm Hierarchical_Trav. A hierarchy may contain multiple roots, i.e., preferences that do not override others (Sect. 3.2). Preferences that are properly overridden ($\sqsubseteq$) by

Algorithm HIERARCHICAL_TRAV

Input: a preference hierarchy $\mathsf{G}_H(\mathcal{V}_H,\,\mathcal{E}_H)$, preference ($p$,$\mathsf{P}$)
Output: a preference hierarchy $\mathsf{G}_H(\mathcal{V}_H,\,\mathcal{E}_H)$
Begin
1. *is_root*=true
2. *RQ*=$\varnothing$
3. **While** $\exists$ root preference $p_r \in \mathcal{V}_H$ not examined {
3.1. *Relationship*=FIND_RELATIONSHIP(p, p_r)
3.2. **If** *Relationship* = 'p is overridden by p_r' {
3.2.1. create edge $e(p, p_r)$ in $\mathcal{E}_H$
3.2.2. unmark p_r }
3.3. **If** *Relationship* = 'p_r is overridden by p' {
3.3.1. is_root=false
3.3.2. **If** $\nexists$ visited edge $e(p_r, *)$ in $\mathcal{E}_H$ {
create edge $e(p_r, p)$ in $\mathcal{E}_H$ }
Else {
Foreach not visited edge $e(p_r, p_i)$ in $\mathcal{E}_H$
add (p_r, p_i) in *RQ*
add $(p_r, -)$ in *RQ*
inserted=false } } }
3.4. **While** *RQ* $<> \varnothing$ {
3.4.1. get head element (p_r, p_i) from *RQ*
3.4.2. **If** p_i=$-$ and *inserted*=false {
create edge $e(p_r, p)$ in $\mathcal{E}_H$
inserted=false }
Else {
Relationship=FIND_RELATIONSHIP(p, p_i)
If *Relationship* = 'p is overridden by p_i' {
remove edge $e(p_r, p_i)$ from $\mathcal{E}_H$
create edge $e(p_r, p)$ in $\mathcal{E}_H$
create edge $e(p, p_i)$ in $\mathcal{E}_H$
inserted=true }
If *Relationship* = 'p_i is overridden by p' {
If $\nexists$ visited edge $e(p_i, *)$ in $\mathcal{E}_H$ {
create edge $e(p_i, p)$ in $\mathcal{E}_H$
inserted=true }
Else {
Foreach not visited edge $e(p_i, p_j)$ in $\mathcal{E}_H$
add (p_i, p_j) in *RQ*
add $(p_i, -)$ in *RQ* } } } }
4. **If** *is_root*=true
4.1 { mark node p as a root }
5. output $\mathsf{G}_H(\mathcal{V}_H,\,\mathcal{E}_H)$
End

p, or vice versa, may be found at different places in the hierarchy, under the same or different roots. To count the above, the algorithm HIERARCHICAL_TRAV starts from each root and performs a breadth-first traversal of the corresponding subgraph. In order to find the $\sqsubseteq$-relationships between the new preference and the preferences in the hierarchy, it searches for the following patterns. Each pattern identifies one or more $\sqsubseteq$-relationship on the basis of $\sqsubset$-relationships.

1. ($p \sqsubset p_r$ and p_r is the root of a subgraph) $\Longrightarrow p \sqsubseteq p_r$
2. ($p_i \sqsubset p$ and p_i is a leaf of a subgraph or all its children are independent with p) $\Longrightarrow p_i \sqsubseteq p$
3. ($p_i \sqsubset p \sqsubset p_j$ and p_i, p_j are nodes of the hierarchy with $p_i \sqsubseteq p_j$) $\Longrightarrow p_i \sqsubseteq p \sqsubseteq p_j$

For instance, the last pattern says that given two preferences p_i and p_j in the hierarchy with p_i being properly overridden by p_j, if the new preference p overrides p_i and is overridden by p_j, then the edge between p_i and p_j is replaced by two edges that essentially place p between the two preferences. This also shows that as the algorithm HIERARCHICAL_TRAV may be called for a sequence of preferences, relationships in the hierarchy are added, modified or enriched under the light of each new preference considered. If a preference is not involved in any relationships, then it becomes a new root in the hierarchy. In principle, the algorithm goes down a subgraph as far as possible from its root and its traversal is guided by two pruning rules:

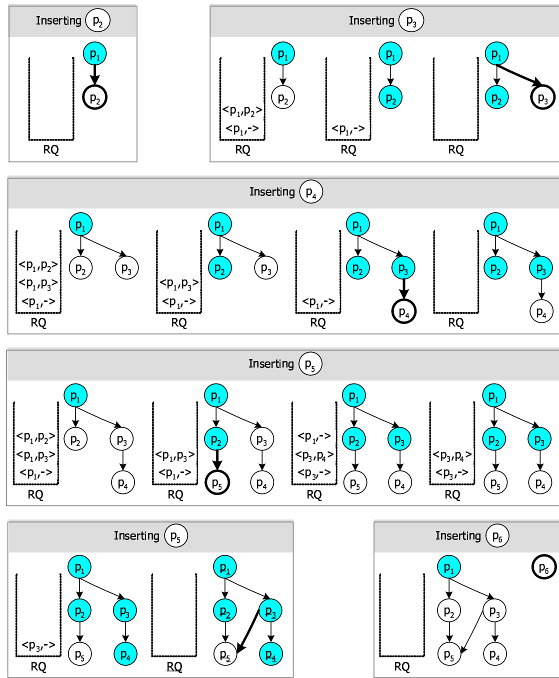


Figure 5: Examples of preference integration.

1. Given a node in the hierarchy, the subgraph starting from this node is not explored if the preference mapping to this node, is independent from or overrides p.
2. Subgraphs with different roots may overlap. Any edge already visited is not visited again because the underlying subgraph is already explored and can be safely pruned.

It is proved that HIERARCHICAL_TRAV is *correct*: it correctly places a preference in a hierarchy by establishing *all* existing $\sqsubseteq$-relationships with the preferences in the hierarchy. (Proof omitted for space considerations.)

The algorithm integrates a preference p, represented as a set of path strings P, into a preference hierarchy $\mathsf{G}_H(\mathcal{V}_H, \mathcal{E}_H)$. A queue RQ keeps edges in the same subgraph to be examined next. These are edges that have not been visited before and are added always in RQ's tail. FIND_RELATIONSHIP is responsible for finding the relationship of each pair of preferences examined (Sect. 4.1). If p is overridden by a root p_r (ln:3.2), then the property of being root is transferred from p_r to p. If p overrides p_r, then the algorithm will search within the underlying subgraph to put p in the right position w.r.t. its relationships to the other preferences in this subgraph. For this purpose, all edges from p_r to its children are added in the queue (ln:3.3.2). The algorithm goes down a subgraph as long as there is a node p_i that is overridden by p (otherwise the underlying part of the subgraph gets pruned.) Then, this node's outgoing edges are added in RQ along with a dummy edge $(p_i, -)$, whose role will be explained shortly. If there are no outgoing edges, then p becomes a child of this node. If p is overridden by p_i, then the algorithm 'breaks' the edge between p_i and its predecessor, and creates two edges, one connecting the predecessor to p and one from p to p_i. The algorithm keeps track whether all children of the same parent are independent from p; in that case, p will become a new child for this node. That is the role of the dummy edge, which is inserted into RQ along with a node's actual outgoing edges. When a dummy edge $(p_r, -)$ is picked from RQ and p has been found independent from p_r's children (i.e., *inserted*=false), then p becomes p_r's new child. Finally, if p does not override any other preference (i.e., *is_root*=true), it becomes a root.

Algorithm EXCLUDE&COMBINE

Input: query Q, a constraint L
a preference hierarchy $\mathsf{G}_H(\mathcal{V}_H, \mathcal{E}_H)$
Output: personalized results R
Begin
1. $R=\varnothing$
2. **Foreach** p_i in $\mathcal{V}_H$ {
2.1. R_i:=$execute_query(Q \wedge p_i)$ }
3. p:=$getnextNodeBottomUp(\mathsf{G}_H)$
4. **While** $\exists p$ {
4.1. **Foreach** $e(p_i, p)$ in $\mathcal{E}_H$ {
4.1.1. R_i:=R_i-R }
4.2. p:=$getnextNodeBottomUp(\mathsf{G}_H)$ }
5. **While** $\exists$ at least L non-empty result sets {
6.1. remove from all result sets the head with the greatest *tid*
6.2. **If** *tid* is found in at least L sets {
6.2.1. $doi(tid) := r(\{d_i | d_i{=}doi(tid \in R_i), \forall R_i \text{ having } tid\})$
6.2.2. add $(tid, doi(tid))$ in R } }
7. output R
End

To illustrate the above, Figure 5 presents the construction of the preference hierarchy depicted in Figure 3. Preferences p_1 to p_6 are presented to the algorithm in that order. Each block in the figure shows the steps required for inserting one preference in the hierarchy. Each step is described by the contents of RQ and the status of the hierarchy, with nodes already examined in color. For instance, the first step for inserting p_3 visits the root p_1, which is overridden by p_3. Hence, its outgoing edges plus a dummy edge are placed in RQ. In the second step, the edge going to p_2 is obtained from RQ and as a result p_2 is visited. This one is independent from p_3, hence the algorithm will not search below this preference. Finally, pulling out the dummy edge $< p_1, - >$ marks the end of p_1'children examination and since p_3 has been found independent from all of them, it is connected to p_1. p_5 provides an example of how the algorithm moves until it establishes all relationships that p_5 participates. The first relationship (i.e., with preference p_2) is found in the second step, but the algorithm goes on exploring the subgraphs starting from p_2's siblings, and ends in discovering the second relationship (i.e., with preference p_3). Finally, observe how p_6 becomes a root.

4.4 Hierarchical Preference Query Answering

Given a query Q and a set of K related preferences that form a hierarchy, the last step of personalization is responsible for returning query results that (a) satisfy at least some of the preferences, i.e., $L \in [1..K]$ preferences, and (b) respect the preference relationships. Relying directly on SQL to capture both requirements can lead to extremely complex, time-consuming queries. The disadvantages of such approaches to query personalization have been studied in the literature (e.g., in [17]), and hence are not discussed any further here. We describe two new approaches.

Algorithm Exclude&Combine. The algorithm is built upon the following intuition. We can execute a number of simple queries. Each query Q_i is a combination of the initial query Q and a preference $p_i (i = 1..K)$ in the current hierarchy and returns a set R_i of results. In this way, as Figure 6 illuminates, we can go from a hierarchy of preferences (Fig. 3) to a "hierarchy of result sets", where nodes and edges have the following meaning: each node R_i represents the results that satisfy preference p_i (without taking into consideration

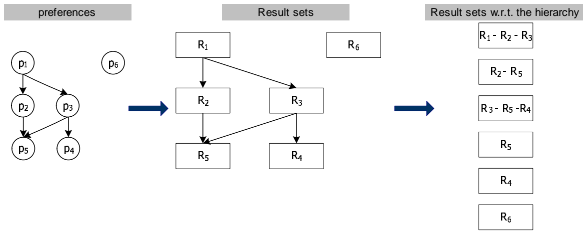


Figure 6: Resultsets w.r.t. a preference hierarchy.

any preference relationships) and each edge from a node R_i to a node R_j implies a difference operation on the result sets of the respective nodes in order to find the results that satisfy p_i but not p_j (i.e., respecting the relationship of p_i and p_j). For instance, taking into consideration the edge from R_3 to R_4 in Figure 6, we understand that R_3-R_4 is the 'actual' set of results for which p_3 should hold because the more specific p_4 does not hold for them. Consequently, we can traverse the hierarchy graph following the edges in their opposite direction, i.e., starting from the leaves and going up to the roots. For each node considered, we remove its results from the result sets of its parent nodes in the hierarchy. In the end, the tuples remaining in each result set will be those that satisfy the respective preference (i.e., the preference that initially generated this set) but not any preference more specific than that (right side of Fig. 6). Then, tuples that occur in at least L sets are those that satisfy at least L preferences, and comprise the final answer.

The algorithm EXCLUDE&COMBINE implements this idea and it is easily proved that it is *correct*, i.e.,: for a query Q and a hierarchy $\mathsf{G}_H(\mathcal{V}_H,\ \mathcal{E}_H)$ of related preferences, it generates all the tuples that satisfy at least L preferences w.r.t. the preference relationships in G_H (proof omitted for the sake of space.) The algorithm proceeds into two steps. (*Exclude*) First it generates the partial result sets, each one satisfying a single preference. Then, it excludes from each set all tuples that satisfy other, more specific, preferences. Each set is ordered on the tuple id *tid*. (*Combine*) As long as there are L non-empty sets, the algorithm removes the greatest *tid*. If this is found in at least L sets, then its degree of interest $doi(tid)$ is computed from the preferences corresponding to these sets and $(tid, doi(t))$ is added in the list R of results, which is kept ordered on the degree of interest.

Algorithm Replicate&Diffuse. This algorithm is based on the idea of visualizing a preference hierarchy as a system of pipes with preference nodes acting as safety valves: When a tuple enters the system at some node (as a result of satisfying the corresponding preference), it rolls down the pipes (edges) starting from this node as long as there is a safety valve that can be "opened". A safety valve will remain closed to a tuple, if the latter satisfies the preference corresponding to the valve, but the valve leads to no other preferences that can be satisfied. Moreover, a tuple satisfying a preference at any level of a hierarchy satisfies its ancestors too. This means that any tuple satisfying at least one preference in the hierarchy will "enter" the system from the hierarchy's roots and roll down following edges from general to more specific preferences, until it is collected by valves mapping to the most specific preferences satisfied by it. The algorithm REPLICATE&DIFFUSE implements this idea in three steps, repeated for each root of a given hierarchy. It creates a set of queries, each one combining the user query Q with a root preference, in order of increasing selectivity. (*Create*) For each root, the algorithm first executes the respective query

Algorithm REPLICATE&DIFFUSE

Input: query Q, a constraint L
a preference hierarchy $\mathsf{G}_H(\mathcal{V}_H,\ \mathcal{E}_H)$

Output: personalized results R

Begin

1. $R = \varnothing$

2. **Foreach** root $p_r \in \mathcal{V}_H$ in order of selectivity {

2.1. $QP = \varnothing$

2.2. $R_r := execute_query(Q \wedge p_r)$

2.3. **Foreach** $tid \in R_r$ {

2.3.1. $execute_query(Q_r(tid))$

2.3.2. update all R_s, $\forall p_s$ following p_r }

2.4. **Foreach** root $p_s \in \mathcal{V}_H$ after and includ. p_r with $R_s <> \varnothing$

2.4.1. add each outgoing edge $e(p_s, p_i) \in \mathcal{E}_H$ in QP

2.5. **While** $QP <> \varnothing$ {

2.5.1. get head element $e(p_i, p_j)$ from QP

2.5.2. **Foreach** $tid \in R_i$

$Temp := execute_query(Q_j(tid))$

If *tid* is found in $Temp$

remove *tid* from R_i

add *tid* in R_j

2.5.3. **If** $R_j <> \varnothing$

add each outgoing edge $e(p_j, p_k) \in \mathcal{E}_H$ in QP }

2.6. **While** $\exists$ at least L non-empty result sets {

2.6.1. remove from all sets the head with the greatest *tid*

2.6.2. **If** *tid* is found in at least L sets {

$doi(tid) := r(\{d_i | d_i = doi(tid \in R_i), \forall R_i$ having $tid)$

add $(tid, doi(tid))$ in R } } }

3. output R

End

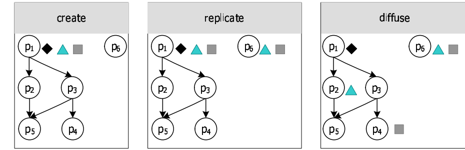


Figure 7: Example of replicate-diffuse steps.

and creates the set of results that satisfy the root preference. (*Replicate*) For each tuple in this set, it finds which root preferences following the current one in order of selectivity are also satisfied, and creates a "replica" of the tuple on the corresponding nodes. (*Diffuse*) Then, it "lets" tuples roll down the hierarchy. In the end, all tuples are found only in nodes corresponding to the most specific, independent preferences they satisfy. Figure 7 illustrates these steps for the root p_1. The algorithm retrieves Q's tuples that satisfy p_1, i.e., three tuples depicted as different shapes. Next, it examines whether any of these satisfy p_6. Assume that the triangle and square tuples do, so they are replicated on this node. Then, all tuples move freely down the hierarchy until they get stopped by some valve. For instance, we find that the square tuple satisfies the independent p_4 and p_6.

The algorithm uses a query Q, a constraint L, and a hierarchy $\mathsf{G}_H(\mathcal{V}_H,\ \mathcal{E}_H)$ of related preferences to generate personalized results for Q. For each root p_r in the hierarchy, it executes a query $Q \wedge p_r$. For each tuple returned by this query with tuple id *tid*, the algorithm executes a parameterized query $Q_r(tid)$, which checks whether *tid* satisfies any other root preference following p_r in order of selectivity (ln:2.3). In this way, *tid* is placed in the result sets of all root preferences (after and including p_r) that it satisfies. Then, the algorithm traverses the hierarchy vertically in order to place the tuples found in the roots in nodes corresponding to independent preferences. A queue QP keeps edges to be examined next, initially containing the outgoing edges from those roots that have non-empty tuple sets

(ln:2.4). New, not visited, edges are added in QP's tail. When an edge $e(p_i, p_j)$ is picked from QP (ln:2.5.1), the algorithm executes a parameterized query $Q_j(tid)$ that checks which of the tuples from p_i's tuple set satisfy p_j. All tuples from R_i that satisfy p_j are moved to R_j. If R_j is non-empty, p_j's outgoing edges, if exist, are added in QP. Once all tuples have been distributed to the right nodes, the algorithm computes the degree of interest of the tuples that satisfy at least L preferences in a similar fashion with the algorithm EXCLUDE&COMBINE, all tuple sets are emptied and the whole process is repeated for the next root preference. It is proved that the strategy that the algorithm follows leads to *correct* answers, i.e., containing all tuples that satisfy at least L preferences w.r.t. the preference relationships.

5. EXPERIMENTS

The novelty of our framework stems from allowing both generic and specific preferences to be explicitly stated in a profile with user-specific, "freely" selected, degrees of interest assigned to them and raising the assumption that preferences can hold independently of each other ("independence assumption"). With this level of expressivity being tempting and found in many real-life scenarios, two questions arise.

- "*Simplicity or expressivity?*" One question is whether the proposed model can have a higher impact than a simpler approach to preference modeling or whether "simple is still beautiful".
- "*Expressivity or performance?*" Personalization of a query using a preference hierarchy requires sophisticated algorithms. Is expressivity achieved at the expense of performance and to what extent? Does this tradeoff justify the use of this framework instead of a simpler one that keeps the "independence assumption"?

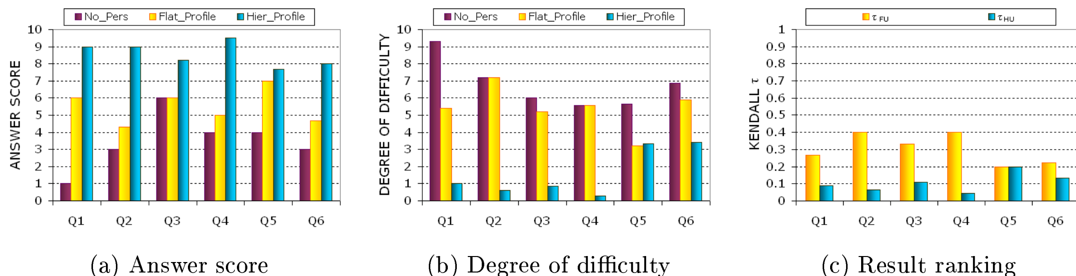
These questions are central in our experimental study. To address the former, we evaluated the preference model through a limited user study. For the latter, we performed an extensive experimental evaluation of our algorithms.

5.1 User Study

We conducted an empirical evaluation of our approach with 11 human subjects with or towards a diploma in computer science. We used a real database containing information about movies from IMDB (www.imdb.com), and we built a web interface that allowed users to manually create their preference profiles and perform searches. In order to gain insight as to the appropriateness of the proposed hierarchical preference model and its benefits for query personalization compared to flat preference representations, each user provided two profiles in the system, one containing only independent preferences (FLAT_PROFILE) following the model presented in [16] and one following the model presented (HIER_PROFILE). An initial test for the appropriateness of the hierarchical model took place during the creation of these profiles. Two interfaces were offered: one for providing simple, independent preferences, and an advanced one for describing more complex preferences. Users were asked to choose one interface for formulating their preferences. 8 of 11 people went with the advanced interface, 2 started with the simple interface and switched to the advanced one and 1 used only the simple interface. The natural selection of the advanced interface was a first indication in favor of the HIER_PROFILE. Then, the users were asked to create a second "view" of their preferences using the other interface. Hence, 10 people had to create a FLAT_PROFILE and 1 had to elaborate his preferences in a HIER_PROFILE. The largest group of users complained for having to "reformulate" their preferences as simpler ones, and expressed their early concerns regarding the system's ability to provide accurate personalization when it relies on these simple, partially correct, profiles. These observations seem to indicate that people tend to trust more a system that captures their preferences more accurately. Finally, the type of hierarchies expected in practice depends on the way user information is (explicitly or implicitly) collected. Also, different people have different types of preferences, depending on their background, their expectations etc. In our experiments, the "full-fledged" hierarchies that could be derived from the hierarchical profiles had an average depth of 3, and the average number of preferences stored per profile was 21.

The following trial was conducted. All subjects were given 4 preselected queries plus 2 of their own choice. Each user submitted these queries three times in arbitrary order. Queries were executed once without personalization (NO_PERS), once using the user's HIER_PROFILE and once using the user's FLAT_PROFILE. The system randomly rotated these options so that the user would not be aware of the query processing performed and hence evaluate query answers unbiased. As parameters for personalization, we chose K to be half of the preferences in a profile and L=1. We ranked results based on the average degree of interest of the preferences satisfied and returned (up to) top-10 results. In the case of NO_PERS, the first 10 results for the query were returned. Users evaluated query answer using two scores measuring [16]: (a) the difficulty to find something interesting, if anything was found at all (DEGREE OF DIFFICULTY) and (b) how satisfactory was the answer (ANSWER SCORE) (both scores in the range [0, 10].) They were also asked to re-rank the tuples returned for each query and put them in an order that they thought closer to their preferences.

Figures 8(a) and 8(b) present the average answer score and degree of difficulty, respectively, per query for each of the three different runs, i.e., NO_PERS, HIER_PROFILE and FLAT_PROFILE. The use of the hierarchical profiles substantially reduces the difficulty to find interesting tuples within an answer and attracts distinctively higher answer scores. Using the flat profiles improves answers (to a lesser degree than HIER_PROFILE). However, we observe that often the improvement is marginal compared to NO_PERS and the degree of difficulty in many cases remains high. Figure 8(c) illuminates the reasons behind this phenomenon. This figure shows how close the ordering of results using the hierarchical profile or the flat profile is to the user's ordering of the same results for each query. There are several standard methods for comparing two lists that are permutations. We used the normalized Kendall tau distance (τ) [14], which lies in the interval [0,1] with 1 indicating maximum disagreement. τ_{FU} compares the list of results based on the FLAT_PROFILE and the same list when re-ordered by the user. τ_{HU} compares the list of results returned using HIER_PROFILE and the same list re-ordered by the user. Figure 8(c) plots the average distances for each query. We observe that when the FLAT_PROFILE was used, users often disagreed with the system-based ordering of results because it did not quite respect their real, more elaborate, preferences. A closer look at the user-ordered results revealed that often 1 to 3 tuples appeared out of order in the sys-



(a) Answer score (b) Degree of difficulty (c) Result ranking

Figure 8: Impact of hierarchical preferences and benefits of query personalization.

tem answers due to the less accurate profiles applied. On the other hand, *more accurate, finer-grained result rankings could be achieved with the hierarchical profiles being more close to user expectations*. The accuracy achieved also depends on the query. For example, for the query $Q5$, none of the profiles had adequate (or different) information for differentiating the output of this query. Finally, the figure brings up another issue: one would expect that since the hierarchical profiles captured the exact user preferences, τ_{HU} would be 0 in all cases. Users still moved tuples in the results for different reasons (e.g., they knew some additional information, such as reviews for certain movies, or while inspecting the results they were able to evaluate them based on properties they had not captured in their profiles.)

5.2 Performance Study

Our approach for evaluating the performance of query personalization using the hierarchical preference model is the following (specific details are given per experiment.) We divide the query personalization process in two main phases: a preference construction phase, which builds a hierarchy of related preferences for a query and a hierarchical preference query answering phase, which generates personalized results using this hierarchy. We study each phase separately and identify the critical factors that affect their performance before discussing the overall performance of personalization. Two parameters play significant roles:

- the number K of selection preferences manipulated
- the number E of $\sqsubseteq$-relationships existing among them

We test our algorithms and we compare them with simpler algorithms that operate on the assumption of independent preferences. We generate synthetic profiles and hierarchies depending on the requirements of each experiment, as we explain in the following subsections, and we build the indexes required for the processing. Times are shown in ms.

5.2.1 *Preference Construction*

This phase involves two tasks: it extracts related preferences from a profile and builds their hierarchy. We want to evaluate the performance of each task. Hence, we measure:

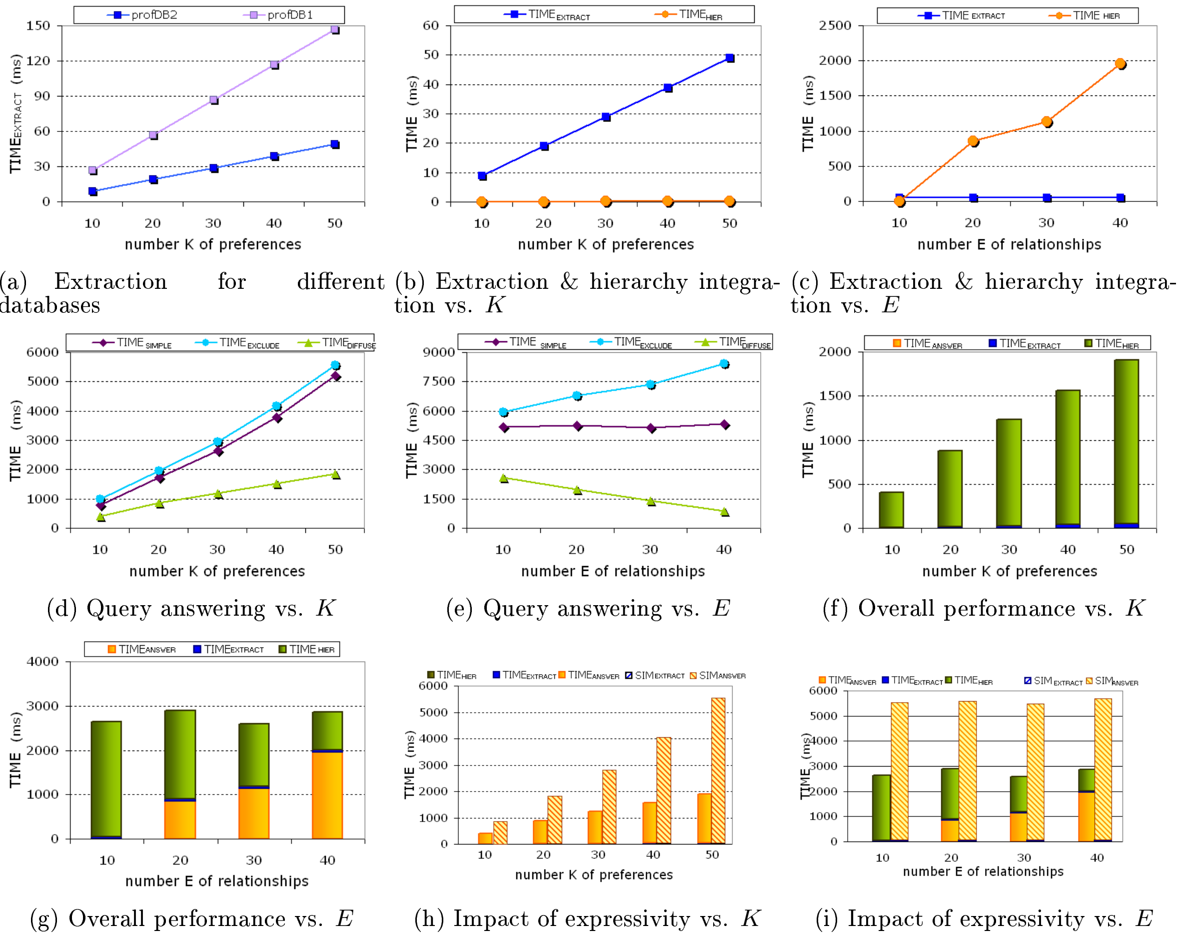
- the total time required to extract K selection preferences from a user profile - Time$_{\mathrm{EXTRACT}}$
- the total time required to find the relationships among K preferences and build their hierarchy, i.e., the total time required by the *Preference-Hierarchy Integration* and *Relationship Finder* modules - Time$_{\mathrm{HIER}}$

Our experiments (not included here due to space constraints) have shown that the time Time$_{\mathrm{HIER}}$ for processing independent preferences, as well as the extra time in Time$_{\mathrm{EXTRACT}}$ for preparing the structures required in preference comparisons, such as representing preferences as trees of path strings, are negligible. This observation allows us to consider that when having only independent preferences, the required execution time for this phase is equal to the time Time$_{\mathrm{EXTRACT}}$, and hence, the overhead from the processing of hierarchical preferences is reflected in Time$_{\mathrm{HIER}}$.

Time$_{\mathrm{EXTRACT}}$ depends on the number K of preferences handled, as Figure 9(a) confirms. However, this task does not simply read preferences stored in a profile but it also builds new preferences from the stored ones. Depending on the database schema and the number of selection preferences defined per relation, it may be able to retrieve K stored selection preferences with a few database accesses or it may need to search in relations further away from the initial query relations and compose preferences out of many stored ones. In order to gain insights into the impact of this phenomenon over Time$_{\mathrm{EXTRACT}}$, we generated the schema of a hypothetic database comprised of 100 relations, each one having 3 attributes, one of which possibly joining this relation to another. Then, we generated two synthetic profile databases of 100 profiles each. Profiles in both databases contained 100 selection preferences each but were generated in a different way: a profile in *profDB1* was generated with the constraint that each attribute of the database could be used in at most one preference, while a profile in *profDB2* was generated with the constraint that each attribute of the database should be used in at most three selection preferences. Figure 9(a) shows Time$_{\mathrm{EXTRACT}}$ as a function of K over all (100) profiles of each database. The difference in execution times can be interpreted as the algorithm's effort to collect K preferences depending on how preferences are distributed over the database. When preferences are sparsely placed (e.g., in *profDB1*), it takes substantially more effort as K increases because it needs to search more in a profile.

In order to measure Time$_{\mathrm{HIER}}$, we built a synthetic hierarchy generator, which takes as inputs the number K of preferences, the number E of preference relationships, and generates a hierarchy with these characteristics, i.e., containing E $\sqsubseteq$-relationships between K preferences, generated in a random way but w.r.t. certain constraints, e.g., defining at most one relationship per pair of preferences. The set of preferences and relationships of a hierarchy are fed into the *Relationship Finder*, which is now in position of helping *Preference-Hierarchy Integration* to build the same hierarchy from scratch. The latter is presented with the set of preferences in random order and is allowed to pose to the *Relationship Finder* only questions regarding $\sqsubset$-relationships.

Figure 9(b) shows the execution times as a function of K assuming E=5. Each point in the figure is the average of execution times for 100 hierarchies with the same characteristics. We observe that the time Time$_{\mathrm{HIER}}$ for the construction of a hierarchy is not greatly affected by K and it is



(a) Extraction for different databases

(b) Extraction & hierarchy integration vs. K

(c) Extraction & hierarchy integration vs. E

(d) Query answering vs. K

(e) Query answering vs. E

(f) Overall performance vs. K

(g) Overall performance vs. E

(h) Impact of expressivity vs. K

(i) Impact of expressivity vs. E

Figure 9: Times for personalizing queries using K preferences organized in a hierarchy with E relationships.

negligible compared with $\text{TIME}_{\text{EXTRACT}}$. On the other hand, Figure 9(c) shows the execution times as a function of E assuming K=50. $\text{TIME}_{\text{EXTRACT}}$ is constant with E. We observe that as the preference hierarchy gets more complicated (i.e., with E increasing), $\text{TIME}_{\text{HIER}}$ deteriorates and finally exceeds $\text{TIME}_{\text{EXTRACT}}$. Hence, we observe that each task's contribution to the total execution time for Preference Construction is different: one task may dominate the other depending on the parameter changing. Ultimately, the overhead from constructing a hierarchy for a query comes not from the number of possibly complex preferences but from the number of relationships among them.

5.2.2 Hierarchical Preference Query Answering

For this phase, we want to evaluate the performance of the two algorithms proposed and the overhead occurred due to the preference hierarchies. Hence, we measure:

- the execution time required by the EXCLUDE&COMBINE algorithm - $\text{TIME}_{\text{EXCLUDE}}$
- the execution time required by the REPLICATE&DIFFUSE algorithm - $\text{TIME}_{\text{DIFFUSE}}$
- the execution time required by a NAÏVE algorithm that works with independent preferences - $\text{TIME}_{\text{SIMPLE}}$

We built NAÏVE as a simple version of EXCLUDE&COMBINE by omitting the exclude step of the latter. NAÏVE executes the set of queries that integrate one preference in the initial query, and combines their results, w.r.t. the constraint that at least L preferences are satisfied, treating the preferences as independent. For this series of experiments, we used the movie database. We generated a set of 50 random queries representing hypothetic user queries, each one containing one relation and one selection on this relation. Note that experiments with other sets of queries with different features, e.g., with one join and one selection, show similar trends and are not discussed for the sake of space. We also generated different sets of hierarchies, each set containing 50 hierarchies with K of preferences and E edges. Each hierarchy was meant to be combined with one query from the above set, and was generated as follows: we first composed K independent selection preferences for our movie database related to the query, with the constraint that they contained only one selection (but any joins required.) Then, we ran an iterative procedure that randomly picked E pairs of preferences and combined them to complex preferences. In essence, at each round, in a pair (p_i, p_j), p_j was replaced by $p_i \wedge p_j$ to form a more specific preference than p_i.

Execution times for this phase depend on three parameters: K, E and L. Due to space constraints, we have selected to show results for K and E, for which we have observed the most significant patterns. We only mention that all three times measured decrease with L increasing, because they process fewer tuples, with $\text{TIME}_{\text{DIFFUSE}}$ being benefited the most, because it processes fewer queries.

Figure 9(d) shows execution times as a function of K with E=5. Each point in the figure is the average of execution

times for the 50 queries combined with their respective hierarchies for the same K and E values. EXCLUDE&COMBINE requires more time than the NAÏVE approach. Since they are both built on the same philosophy, the overhead observed is due to the additional actions required to sort out results w.r.t. preference relationships. The surprise comes from the time $\text{TIME}_{\text{DIFFUSE}}$, which is substantially lower even from $\text{TIME}_{\text{SIMPLE}}$. The "secret" of REPLICATE&DIFFUSE lies in its systematic approach to processing queries and results: in a nutshell, it exploits query selectivity, preference relationships, and parameterized queries to execute as few queries as possible and process a small number of tuples close to the real number of results returned. Figure 9(e) shows times as a function of E with K=50. While $\text{TIME}_{\text{EXCLUDE}}$ deteriorates when more preference relationships need to be resolved, $\text{TIME}_{\text{DIFFUSE}}$ is actually benefited because the total number of queries executed is lower either due to fewer queries of the type $Q \wedge p_i$ executed or to fewer parameterized ones. Hence, REPLICATE&DIFFUSE for generating results based on preference hierarchies exhibits a better behavior in contrast to a naïve approach that works with independent preferences because it exploits preference relationships.

Overall, our results so far indicate that allowing complex preferences in query personalization may make preference construction more expensive but benefit query answering. How these phenomena shape the total execution time?

5.2.3 Overall Performance

To complete the picture regarding the efficiency trade-offs, we compare the contribution of all parts in the total processing time of a query, and we plot the time $\text{TIME}_{\text{EXTRACT}}$ required to extract K selection preferences, the time $\text{TIME}_{\text{HIER}}$ required to find the relationships among K preferences and build the hierarchy and the time $\text{TIME}_{\text{ANSWER}}$ to generate the personalized results. For the latter, we consider the time $\text{TIME}_{\text{DIFFUSE}}$, since REPLICATE&DIFFUSE has been shown to be the most efficient. Figure 9(f) shows the execution times as a function of K for E=5 and Figure 9(g) shows the execution times as a function of E for K=50. When K increases the execution time of the hierarchical preference query answering phase shapes the overall performance. On the other hand, when E increases, we witness the effect of two opposite forces: decreasing time for generating results tends to "compensate" to a certain degree for the increase in the time required for building the hierarchy. Thus, the whole personalization process behaves in a more balanced way. Finally, Figures 9(h) and 9(i) compare the efficiency of personalization using hierarchies with personalization using only simple, independent preferences. $\text{SIM}_{\text{EXTRACT}}$ and $\text{SIM}_{\text{ANSWER}}$ are the execution times of the algorithms when extracting independent preferences related to a query and when generating the personalized answer, respectively. Overall, taking advantage of preference relationships can help the algorithms adapt more smoothly to changes to K or E. *Hence, performance is not sacrificed for expressivity but it can actually benefit from it.*

6. CONCLUSIONS AND FUTURE WORK

We have presented a framework for the explicit formulation of preferences at varying levels of granularity. We have introduced the concept of a *preference hierarchy*, where each node refers to a subclass of the entities that its parent refers to, and whenever they both apply, more specific preferences override more generic ones. We have provided algorithms for query personalization based on preference hierarchies. Finally, we have evaluated the framework through a limited user study and performed an extensive experimental evaluation of our algorithms. The overall message is that more accurate, finer-grained result rankings can be achieved with the proposed hierarchical preference model without trading off performance. In addition, the work presented here comes with an interesting research agenda of related issues, such as the efficient management of complex preferences, e.g., by exploiting preference commonalities and access patterns in the same or different profiles, designing an intuitive GUI that facilitates defining and editing preferences, and so forth. On a parallel line of research, we work on analyzing query logs from a portal in order to build community profiles represented as hierarchies of preferences.

7. REFERENCES

[1] Y. S. A. Aho and J. D. Ullman. Equivalence of relational expressions. *SIAM J. of Computing*, 8(2):218–246, 1979.

[2] R. Agrawal and E. Wimmers. A framework for expressing and combining preferences. In *SIGMOD*, 2000.

[3] M. Balabanovic and Y. Shoham. Fab: Content-based, collaborative recommendation. *CACM*, 40(3):66–72, 1997.

[4] W.-T. Balke, U. Guntzer, and C. Lofi. User interaction support for incremental refinement of preference-based queries. In *IEEE RCIS*, pages 511–523, 2007.

[5] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[6] C. Boutilier, R. Brafman, H. Hoos, and D. Poole. Reasoning with conditional ceteris paribus preference statements. In *UAI*, 2000.

[7] C. Chekuri and A. Rajaraman. Conjunctive quer containment revisited. In *ICDT*, 1997.

[8] J. Chomicki. Preference formulas in relational queries. *ACM TODS*, 28(4):427–466, 2003.

[9] D. Conte, P. Foggia, C. Sansone, and M. Vento. 30 years of graph matching in pattern recognition. *Int'l J. of Pattern Recognition and Artificial Intelligence*, 18(3):265–298, 2004.

[10] P. Fishburn. Preference structures and their numerical representations. *Theor. Comp. Sci.*, 217:359–383, 1999.

[11] S. Holland, M. Ester, and W. Kiessling. Preference mining: A novel approach on mining user preferences for personalized applications. In *PKDD*, pages 204–216, 2003.

[12] S. Holland and W. Kiessling. Situated preferences and preference repositories for personalized database applications. In *ER*, pages 511–523, 2004.

[13] T. Joachims, D. Freitag, and T. Mitchell. Webwatcher: a tour guide for the world wide web. In *IJCAI*, 1997.

[14] M. Kendall and J. D. Gibbons. *Rank Correlation Methods.* Edward Arnold, London, 1990.

[15] W. Kiessling and W. Kostler. Foundations of preferences in database systems. In *VLDB*, 2002.

[16] G. Koutrika and Y. Ioannidis. Personalization of queries in database systems. In *ICDE*.

[17] G. Koutrika and Y. Ioannidis. Personalized queries under a generalized preference model. In *ICDE*, 2005.

[18] M. Lacroix and P. Lavency. Preferences: Putting more knowledge into queries. In *VLDB*, pages 217–225, 1987.

[19] D. Shasha, J. Wang, H. Shan, and K. Zhang. ATreeGrep: Approximate searching in unordered trees. In *SSDBM*, pages 168–177, 2002.

[20] K. Stefanidis, E. Pitoura, and P. Vassiliadis. Adding context to preferences. In *ICDE*, 2007.

[21] Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou. Branch-and-bound processing of ranked queries. *Inf. Syst.*, 32(3):424–445, 2007.

[22] M. J. Zaki. Efficiently mining frequent trees in a forest. *Inf. Syst.*, 17(8):1021 – 1035, 2005.