Management of User Preferences in Data Intensive Applications

Riccardo Torlone[1] and Paolo Ciaccia[2]

[1] Dip. di Informatica e Automazione, Università Roma Tre
Via della Vasca Navale, 79 - 00146 Roma, Italy
torlone@dia.uniroma3.it
[2] DEIS — CSITE–CNR, Università di Bologna
Viale Risorgimento, 2 - 40136 Bologna, Italy
pciaccia@deis.unibo.it

Abstract. The management of user preferences is becoming a fundamental ingredient of modern Web-based data-intensive applications, in which information filtering is crucial to reduce the volume of data presented to the user. However, though deriving and modeling user preferences has been largely studied in recent years, there is still a need for practical methods to efficiently incorporate preferences in actual systems. In this paper we consider the qualitative approach to user preferences in which a binary preference relation is defined among objects and a special operator (called Best) is used to extract relevant data according to the preference relation. In this framework, we propose and study a special index structure, called β-tree, which can be used for a rapid evaluation of the Best operator. We then present a number of practical algorithms for the efficient maintenance of β-trees in front of database updates and discuss some relevant implementation issues.

1 Introduction

The continued explosion in the amount of content and the number of information sources available online is making the need for effective personalized content delivery more crucial. This has resulted in a renewed interest in personalization as a fundamental tool for Web-based data intensive applications. Personalization can be defined as any action that tailors the interaction with an information system to a particular user, or set of users. The interaction can be something as casual as browsing a Web site or as (economically) significant as trading stocks or purchasing a car. The actions can range from simply making the presentation more pleasing to anticipating the needs of a user and providing customized and relevant information. In this framework, one of the most challenging problem is the management of *user preferences*, that is, representing in the most appropriate way the specific interests of the user (or of a class of users) and deliver, according to them, only the most appropriate information.

In the database field, the problem of expressing and managing user preferences has received growing attention in the last few years [1–5]. A recent and

very general approach relies on *qualitative* preference relations [2, 3, 6]. They are simple binary relations between tuples: each pair of tuples in a qualitative preference relation specifies the preference of one tuple over another one. This approach includes, as a proper sub-case, *quantitative* preferences, which associate a score of preference with each tuple and therefore require, the definition of a linear order over them. Preference queries make use of special operators defined with respect to a given set of preferences. For instance, the *skyline* operator [2] returns all the tuples which are not *dominated* by any other tuple, where t_1 dominates t_2 if t_1 is at least as good as t_2 in all dimensions and better than t_2 in at least one dimension. As an example, hotel t_1 might be considered to dominate hotel t_2 if t_1 is cheaper than t_2 and both have the same rating. Along this line, Chomicki [3] has recently considered the case where formulas used to compare the tuple dimensions are restricted to use only built-in predicates, and predicates can be freely combined with logical connectives. He also proposed an operator, called *winnow*, for the retrieval of tuples according to these formulas.

In this paper we consider qualitative preferences and, for the sake of generality, disregard the orthogonal problems of how preferences are expressed (that is, in which language) and in which way they are collected (that is, defined explicitly by the user or derived implicitly from, e.g., click-streams). With this in mind, we consider the *Best* operator (a variant of the winnow) that selects the $n-$th ranked set of tuples in terms of preference [7, 8]. In particular, unlike previous works, we focus on the case where not only the highest ranked tuples are to be computed, but possibly also tuples with a lower rank are to be returned. In [7, 8] we studied general properties of the Best operator and presented a basic algorithm for its computation. Building on these results, we illustrate in this paper a special index structure, called *β-tree*, that can be used for a fast evaluation of Best queries. A *β-tree* has limited size since it only stores the information about user preferences that is strictly needed for computing Best queries. To our knowledge, this is the first proposal of index structure for preference queries. We also consider the problem of maintenance of this structure in front of database updates and propose efficient algorithms that solve this problem. These results provide the basis for the development of a practical preference systems for *dynamic* data intensive applications.

The paper is organized as follows. In Section 2 we introduce some basic notions about preferences and querying with preferences. In Section 3 we illustrate and investigate the notion of β-tree and, in Section 4 we propose a number of algorithms for its maintenance. Then, in Section 5 we discuss some implementation issues and finally, in Section 6, we draw some conclusions.

2 Preferences in Relational Databases

In this section we briefly recall the notion of user (qualitative) preference [3] and present the notion of reduced preference graph. For simplicity we refer to the relational model but the approach can be easily extended to more involved data models.

2.1 Qualitative Preferences

Let $R(X)$ be a *relation scheme*, where R is the *name* of the relation and $X = A_1, \ldots, A_k$ is a set of *attributes* each of which has associated a set of values D_i called the *domain* of A_i. User preferences over $R(X)$ can be naturally expressed by a collection of pairs of tuples over $R(X)$: each pair specifies the preference of one tuple over another one. These are called *qualitative* preferences [3].

Definition 1 (Preference relation). *A* (qualitative) preference relation $\succ$ *over a relational scheme* $R(X)$ *is a binary relation over* $\prod_{i=1}^{k} D_i$. *We denote a single preference* $(t_1, t_2) \in \succ$ *by* $t_1 \succ t_2$ *and say that* t_1 is preferable *to* t_2.

Example 1. Consider the following set of tuples.

	Make	Model	Color	Price
t_1	BMW	330	Black	30K
t_2	Ford	Escort	White	20K
t_3	Toyota	Corolla	Silver	15K
t_4	Ferrari	360	Red	100K

A set of qualitative preferences over them, possibly expressing the interest of a potential customer, can be the following: $t_1 \succ t_2$, $t_3 \succ t_2$, $t_4 \succ t_1$ and $t_3 \succ t_1$.

A notion of *indifference* [3] can be naturally derived from the definition above.

Definition 2 (Indifference). *Given a qualitative preference relation* $\succ$ *over a scheme* $R(X)$, *an* indifference relation $\|$ *over* $R(X)$ *is defined as follows:* $t_1 \| t_2$ *if neither* $t_1 \succ t_2$ *nor* $t_2 \succ t_1$; *if* $t_1 \| t_2$ *we say that* t_1 is indifferent to t_2.

A preference relation can be naturally represented by means of a directed graph $G_\succ$ that we call *preference graph*. In this graph, nodes correspond to tuples and there is an edge from a node t_1 to a node t_2 if $t_1 \succ t_2$. For instance, the preference graph for Example 1 is reported in Figure 1.

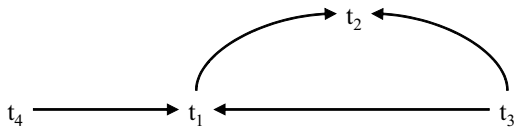

Fig. 1. A preference graph

Several properties of preference relations can be expressed and studied in terms of properties of the corresponding graph and for this reason in the following we will often switch from a preference relation to its graph representation.

Note that, according to the above definition, a qualitative preference is in general an infinite relation. In real application however, we are interested into the restriction of a preference relation to actual data included into a repository.

Thus, given an instance r be over a scheme $R(X)$, that is, a finite set of tuples over $R(X)$, and a preference relation $\succ$ over $R(X)$, we will denote by $\succ_r$ the *restriction* of $\succ$ to r, that is, the subset $\succ_r$ of $\succ$ such that $t_1 \succ_r t_2$ if and only if $t_1 \succ t_2$ and both t_1 and t_2 are in r.

2.2 Properties of Preferences

Usually, in many applications, a preference relation $\succ$ is at least a (strict) partial order [3]. This means that $\succ$ is irreflexive (we never have $t \not\succ t$), asymmetric (we never have both $t_1 \succ t_2$ and $t_2 \succ t_1$), and transitive (if $t_1 \succ t_2$ and $t_2 \succ t_3$ then also $t_1 \succ t_3$). When a preference relation is a partial order, its graphical representation requires a reduced number of edges, since preferences that can be derived by transitivity do not need to be represented. We then introduce the following notion.

Definition 3 (Reduced Preference Graph (RPG)). *Let $\succ$ be a preference relation over $R(X)$ that is a partial order and let r be a relation over $R(X)$. The reduced preference graph (RPG) of $\succ_r$ is the directed graph $\overline{G}_{\succ_r} = (N, E)$ defined as follows:*

1. *There is a node in N for each tuple $t \in r$ plus a special node $t_\top$;*
2. *There is an edge in $G_\succ$ for each pair of tuples $t_1, t_2 \in r$ such that $t_1 \succ t_2$ but no tuple $t' \in r$ satisfies $t_1 \succ t' \succ t_2$;*
3. *There is an edge from $t_\top$ to each node $t \in r$ such that does not exist $t' \in r$ with $t' \succ t$.*

The node $t_\top$ is added in order to have a unique source in $\overline{G}_{\succ_r}$, a technical feature that will be useful in the following. It is easy to show that given any preference relation $\succ$ its RPG $\overline{G}_\succ$ is unique and can be obtained as the transitive opening of $G_\succ$ (an operation that deletes from the graph all the edges that can be derived by transitivity).

Example 2. Figure 2 gives an example of a preference graph and its reduced version.

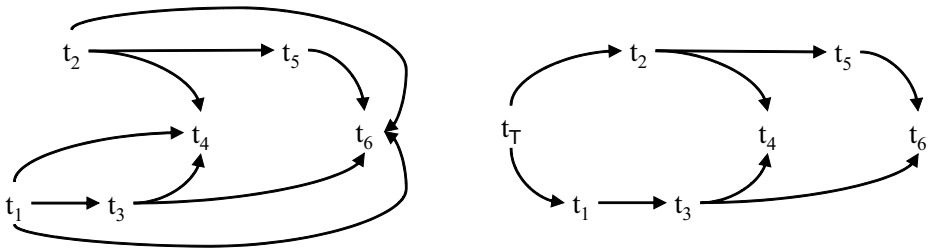

Fig. 2. A preference graph and its reduced version

2.3 Querying with Preferences

In order to smoothly embed preferences in queries, we consider a special operator that, combined with the standard operators of relational algebra, can be used to specify queries over a database with preferences. This operator is called *Best* (denoted by $\beta^n_\succ$) and selects the $n-$th ranked set of tuples in terms of preference (for $n = 1$ we obtain the best tuples in absolute terms).

Definition 4 (Best operator). *Let r be a relation over a scheme $R(X)$ and let $\succ$ be a qualitative preference relation over $R(X)$. The* Best operator $\beta^n_\succ$ of rank $n > 0$ *is defined as follows:*

- $\beta^1_\succ(r) = \{t \in r \mid \nexists t' \in r, t' \succ t\}$
- $\beta^{n+1}_\succ = \beta^1_\succ(r - \bigcup_{i=1}^{n} \beta^i_\succ(r))$

$\beta^1_\succ(r)$ returns all the tuples t of a relation r for which there is no tuple in r better than t according to $\succ$. If the user is not satisfied from the basic result, this operation can be applied iteratively: at each step it returns the best tuples of r, excluding the tuples retrieved in previous steps.

Example 3. Consider the preferences represented by the graph in Figure 1. Then we have $\beta^1_\succ(r) = \{t_4, t_3\}$, $\beta^2_\succ(r) = \{t_1\}$ and $\beta^3_\succ(r) = \{t_2\}$.

There is a strong relationship between the topology of the reduced preference graph associated with a preference relation and the result of the Best operator. In particular, in [8] we have shown the following result.

Lemma 1. *Given a preference relation $\succ$ and its RPG $\overline{G}_\succ$ (1) $\beta^1_\succ(r)$ returns the children of $t_\top$, and (2) for every $t \in r$, $t \in \beta^k_\succ(r)$ where k is the length of the longest path in $\overline{G}_\succ$ from $t_\top$ to t.*

Example 4. Let us consider the RPG in Figure 2. According to Lemma 1 we have that $t_4 \in \beta^3_\succ(r)$ since the longest path from $t_\top$ to t_4 in the RPG has length 3. Note that there is indeed another path from $t_\top$ to t_4 whose length is 2.

In [8] we have proposed the *Beta Algorithm* for the computation of the Best operator. The Beta algorithm is composed by a number of *phases*, one for each iteration of the Best operator. In turn, each phase consists of a number of *scans* over a set C_i of *candidate tuples* which might belong to the output Out_i of the $i-$th phase. Each scan identifies one tuple to be inserted into Out_i and generates, for each examined tuple t, the set of all tuples $D^t_\succ$ dominated by t that have been identified in the scan. The algorithm returns two collections of sets of tuples. The former contains the results of the various iterations of the Best operator: $Out_i = \beta^i_\succ(r)$. The latter contains, for each tuple t examined by the algorithm, the set $D^t_\succ$ of tuples that have been found to be dominated by t during its execution.

3 β-trees

In this section we present and investigate a special data structure, called β-tree, that can be built while running the Beta algorithm and can be later used to compute very efficiently the Best operator.

3.1 Definition of β-tree

Lemma 1 suggests that in order compute the Best operator over a set of tuples with preferences we actually need to know just a portion of the reduced preference graph. This is the idea underlying the notion of β-tree.

Definition 5 (β-tree). *Let $\succ$ be a qualitative preference over a scheme $R(X)$ and let r be a relation over $R(X)$. A β-tree $\mathcal{T} = (N, E)$ over $\succ_r$ is any tree embedded in RPG $\overline{G}_{\succ_r} = (N, E')$ ($E \subseteq E'$) such that, for each $t \in r$, the length of the path from the root of $\mathcal{T}$ to n equals the length of the longest path from the node $t_\top$ of $\overline{G}_{\succ_r}$ to t.*

An important observation is that, by definition, the size of a β-tree is linear in the cardinality of the relation, thus possibly much less than the size of the reduced preference graph.

Example 5. A β-tree for the RPG in Figure 2 is reported in Figure 3. Note that another different β-tree can be built from the same RPG. In such β-tree t_6 is the child of t_5 instead of t_3.

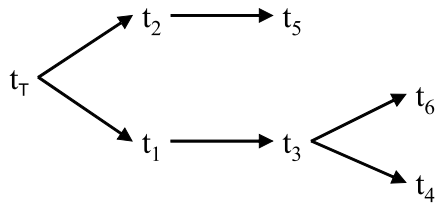

Fig. 3. A β-tree for the RPG in Figure 2

3.2 Construction of a β-tree

Interestingly, it turns out that the Beta algorithm can be used to build a correct β-tree. More precisely, let T_r be the tree defined as follows:

1. T_r has a node t for each $t \in r$ plus a special node $t_\top$;
2. there is an edge from t_1 and t_2 in T_r if, at the end of the basic algorithm, $t_2 \in D^{t_1}_{\succ}$; and
3. there is an edge from $t_\top$ to t in T_r if t has no other incoming edge.

Since the sets $D_{\succ}^{t_i}$ are pairwise disjoint, it is guaranteed that the resulting structure is indeed a tree.

Example 6. Let $r = \{t_1, t_2, t_3, t_4, t_5, t_6\}$ and assume we have the set of preferences represented by the RPG reported in Figure 2. By running the basic algorithm on this input we obtain the following sets of dominated tuples: $D_{\succ}^{t_1} = \{t_3\}$, $D_{\succ}^{t_2} = \{t_5\}$, $D_{\succ}^{t_3} = \{t_4, t_6\}$. From them, we obtain exactly the β-tree reported in Figure 3.

We recall that the *level* of a node n in a tree T is the length of the (unique) path from the root of T to n. Let $L_i(T)$ denote the set of nodes of T at level i (hence, $L_0(T)$ only contains the root of T). By construction and by Lemma 1 we can easily show the following.

Lemma 2. *Let r be a relation over a scheme $R(X)$, $\succ$ be a preference relation over $R(X)$ and T_r the tree built as above. Then:*

1. *T_r is a β-tree over $\succ_r$, and*
2. *$\beta_{\succ}^i(r) = L_i(T_r)$, for each $i > 0$.*

4 Management of a β-tree

4.1 Tuple insertion

Assume we have a preference relation $\succ$ over a relational scheme R and let T_r the corresponding β-tree built as described in Section 3.1. Assume now that we need to insert a new tupe t to r; we have that $\succ_r \subseteq \succ_{r\cup\{t\}} \subseteq \succ$ and thus T_r need to be modified accordingly. The effect of the insertion of a new tuple on the β-tree T_r does not consist in a simple node addition but, in general, to a restructuring of T_r. Regarding that, a number of preliminary results can be stated.

Lemma 3. *Let t be a tuple to be inserted into a relation r over a scheme $R(X)$, $\succ$ a preference relation over $R(X)$ and T_r be a β-tree over $\succ_r$.*

1. *if there is a tuple t' at level $k-1$ of T_r such that $t' \succ t$ and there is no tuple t'' at level k of T_r such that $t'' \succ t$ then $t \in \beta_{\succ}^k(r \cup \{t\})$, and*
2. *if there is a tuple t' at level k of T_r such that $t' \notin \beta_{\succ}^k(r \cup \{t\})$ then $t' \in \beta_{\succ}^{k+1}(r \cup \{t\})$.*

Point (1) specifies a simple way to find the correct level of T_r to which t has to be inserted: at level k of T_r as a child of any tuple t' at level $k-1$ satisfying the specified property. Conversely, point (2) suggest how T_r has to be modified to reflect the insertion of t into T_r, since it states that if a tuple at level k of the β-tree is no more returned at the k-th iteration of the Best operators as effect of the insertion of a new tuple, it has to be moved exactly one level down in T_r.

A method for updating T_r to take into account the insertion of t in r can be therefore divided into two main phases: (1) the identification the correct level k of T_r in which t has to be inserted, and (2) the evaluation of the effect of the insertion of t on the higher levels l ($l > k$) of T_r. The basic ideas that will be used to speed up the computation are (1) moving entire subtrees of T_r instead of single tuples and (2) restricting the search space as much as possible.

First phase The first phase proceeds by inspecting T_r level by level, starting from level 1 and initializing the *parent tuple* of t to $t_\top$. The scan of level i ($i \geq 1$) proceeds as follows.

1. We compare t with a tuple t' in $L_i(T_r)$; the following cases can occur:
   (a) if $t \parallel t'$ no operation is performed;
   (b) if $t \succ t'$ we perform the following operations: (i) we put t' into the (initially empty) set D_i of *declassed* tuples of level i; (ii) we remove from T_r the whole tree $T_{t'}$ rooted at t' and attach $T_{t'}$ as a subtree of t; we construct in this way a tree having t as root that will be denoted by T_t;
   (c) if $t' \succ t$ the tuple t' becomes the parent tuple of t and the scan stops.
2. The scan of of level i stops when either case (c) occurs or all the tuples in $L_i(T_r)$ have been examined:
   (a) in the first case we proceed by inspecting in the same way level $i+1$ of T_r; if i is the last level the first phase is concluded;
   (b) in the second case we just record the parent tuple of t and this concludes the first phase.

Second phase Assume that the first phase is concluded at the end of the inspection of level k. The second phase of the method consists in evaluating the effect of the insertion of t to the levels $l > k$ of T_r. This is done by moving subtrees of T_r to T_t as follows.

1. For each tuple t_d in the set D_k of declassed tuples of level k, we compare t_d with all the tuples at level $k+1$ of T_r. Note that not all the original tuples in $L_{k+1}(T_r)$ will be examined since some of them could have been removed from T_r and inserted into T_t.
2. Only two cases can occur by comparing t_d with a tuple t' in $L_{k+1}(T_r)$: either $t_d \parallel t'$ or $t_d \succ t'$. In the first case no operation is performed; in the second case we proceed as follows: (i) we put t' into the (initially empty) set D_{k+1} of *declassed* tuples of level $k+1$, and (ii) we remove from T_r the whole tree $T_{t'}$ rooted at t' and attach it in T_t as a subtree of t_d.
3. If at the end of the scan of level l ($l > k$) a new set D_l of declassed tuples is generated we proceed in the same way with level $l+1$ until either no declassed tuple is generated or we reach the last level of T_r.
4. The last step of the method simply consists in attaching T_t to T_r as a subtree of the parent tuple of t.

Example 7. Assume that we want to insert t_7 into the relation of Example 6 whose β-tree is reported in Figure 3. Assume also that $t_2 \succ t_7$ and $t_7 \succ t_5$. By inspecting the first level we find that $t_2 \succ t_7$ and so t_2 becomes the parent tuple. At the second level we have no tuple preferable to t_7 but t_5 is declassed, since $t_7 \succ t_5$, and is attached as a subtree of t_7 (Figure 4.a). This concludes the first phase. Now we need to compare the declassed tuple t_5 with the tuples at level 3 of T_r (t_6 and t_4). We have that $t_5 \succ t_6$ (see the RPG in Figure 2) and so t_6 is declassed and is inserted into T_{t_7} as the child of t_5 (Figure 4.b). Conversely,

$t_5 \| t_4$ and so t_4 is not declassed. Since this is the last level we can attach T_{t_6} as a subtree of t_2 (the parent tuple): the algorithm terminates and we obtain the β-tree reported in Figure 4.c.

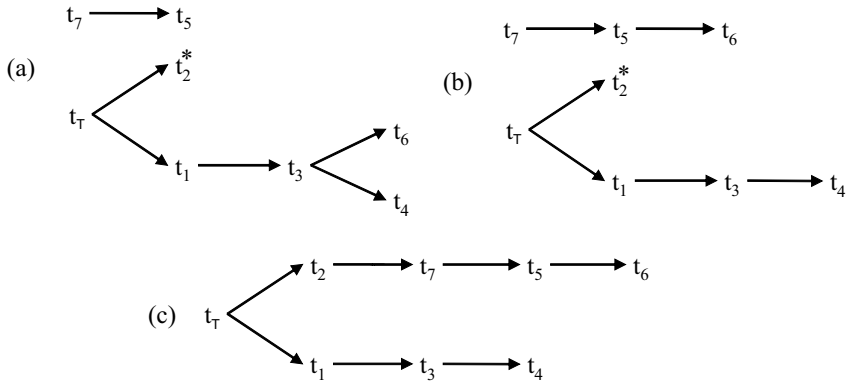

Fig. 4. The execution of the insertion algorithm described in Example 7

Theorem 1. *The insertion algorithm produces a correct β-tree and requires quadratic time in the worst case.*

We point out that while the proposed technique requires no more than $O(n^2)$ comparisons in the worst case (a rather degenerate situation in which all the tuple are distributed along only two levels of the β-tree), it exhibits a very good performance in the average case.

4.2 Tuple deletion

Assume now that we need to delete a tuple t from r: we have that $\succ_{r-\{t\}} \subseteq \succ_r \subseteq \succ$. Again, this update can require a quite involved restructuring of T_r, but we can proceed very rapidly by operating over entire subtrees of T_r. The following result can be easily shown.

Lemma 4. *Let t be a tuple to be deleted from a relation r over a scheme $R(X)$, $\succ$ be a preference relation over $R(X)$ and T_r be a β-tree over $\succ_r$. Then, if there is a tuple t' at level k of T_r such that $t' \notin \beta^k_{\succ}(r \cup \{t\})$ then $t' \in \beta^{k-1}_{\succ}(r \cup \{t\})$.*

This lemma suggests that suggest if a tuple at level k of the β-tree is no more returned at the k-th iteration of the Best operators as effect of the deletion of a tuple, then it has to be promoted of exactly one level in T_r.

Also in this case we can divide the process into two phases: (1) the modification of the β-tree around the level k that includes t, and (2) the evaluation of the effect of the deletion of t on the higher levels l ($l > k$) of the β-tree. Again, we can speed up the computation by operating over entire subtrees.

First phase Let t be at level k of T_r and $\hat{t}$ be the parent of t. The children of t in T_r are the tuples which are *candidate for promotion at level* k . We then proceed by inspecting these tuples as follows.

1. We put the children of t into the set C_k of the candidates for promotion at level k and then delete t from T_r.
2. For each tuple t' in C_k, we compare t' with a tuple t'' at level k of T_r.
   - if $t'' \succ t'$ we attach the whole tree rooted at t' as a subtree of t''.
   - if $t'' \parallel t'$ we proceed by comparing t' with the other tuples at level k of T_r.
3. If we do not find any tuple $t'' \succ t'$ at level k of T_r we attach t' as a child of $\hat{t}$ (the parent of t) and put the children of t' into the set C_{k+1} of the candidate for promotion at level $k+1$.

Second phase The second phase proceeds similarly to the first phase by operating over the set of tuples that are candidate for promotion.

1. For each tuple t' in the set C_l of candidate for promotion at level l (initially $l = k+1$), we compare t' with a tuple t'' at level l of T_r.
   - if $t'' \succ t'$ we attach the whole tree rooted at t' as a subtree of t''.
   - if $t'' \parallel t'$ we proceed by comparing t' with the other tuples at level l of T_r.
2. If we do not find any tuple $t'' \succ t'$ at level l of T_r we just insert the children of t' into the set C_{l+1} of the candidate for promotion at level $l+1$.

This procedure is iterated until no candidate for promotion is generated or we reach the last level of T_r.

Example 8. Assume that we want to delete t_1 from the relation of Example 6 whose corresponding β-tree is reported in Figure 3 (note that t_1 is at level 1). Then, t_3 is the only child of t_1 and therefore it is candidate for promotion at level 1. Hence, after the deletion of t_1, we have to compare t_3 with t_2 (the only tuple at level 1). Since $t_3 \parallel t_2$, t_3 is attached as a child of $t_\top$ (the parent of t_1) and both t_4 and t_6 (the children of t_3) become the candidate for promotion at level 2. We now have to compare t_4 and t_6 with t_5 (the only tuple at level 2). Since $t_5 \parallel t_4$, t_4 remains a child of t_3 (and so it is actually promoted) Conversely, we have $t_5 \succ t_6$ (see the RPG in Figure 2) and so t_6 is attached as a subtree of t_5. No candidate for promotion are generated in this step and so the algorithm terminates producing β-tree reported in Figure 5. Note that among the candidate for promotion only t_3 and t_4 have been promoted.

Theorem 2. *The deletion algorithm produces a correct β-tree and it requires quadratic time in the worst case.*

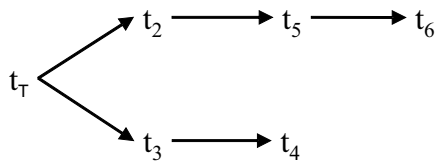

Fig. 5. The β-tree produced as described in Example 8

5 Implementation issues

To verify the effectiveness and the efficiency of our approach, we have started an implementation of a system for the management of preference in a relational database based on β-trees. In this section we briefly discuss some implementation choices done in the realization of the system.

The system makes use of the XXL (eXtensible and fleXible Library) toolkit [9]: a flexible platform independent Java-library that provides a powerful collection of generic index-structures, query operators and algorithms facilitating the performance evaluation of new query processing techniques. The rationale under this choice is that XXL provides many building blocks suitable to implement query processing algorithms. In particular, all operators known from the relational algebra like join or selection are currently implemented in the package. Moreover, XXL is easily extensible: this allows a rapid development of a new relational algebra operator (like the Best) that can be then freely combined with the traditional ones as well as novel database index structures.

The β-tree has been implemented by means of a special hash structure. Specifically, given a β-tree T_r and an hash function h, for each pair of tuples $t, t' \in r$ such that there is an edge from t to t' in T_r, the tid (tuple identifier) τ' of t' is stored in the bucket of address $h(\tau)$, where τ is the tid of t. It follows that all the children of a tuple in T_r are stored in the same bucket. Collisions are taken into account by associating with each tid stored in a bucket the tid of the parent tuple. The advantage of using such an hash structure is that the β-tree can be traversed very efficiently and the tree restructuring required by update algorithms correspond in many cases to simple modifications of the hash table.

Another implementation choice has been that of storing and maintaining only a limited number k of levels of a β-tree (usually $2 \leq k \leq 10$ depending on the cardinality of the levels). The rational under this choice is twofold. From a conceptual point of view, users are usually satisfied by just the first iterations of the Best operator and it would be therefore useless to store deeper levels of a β-tree. From a practical point of view we obtain in this way a very compact representation of a β-tree that can managed very efficiently. With this approach, the size of a β-tree usually decreases over time since deletions do not promote tuples from levels greater than k and insertion drop tuples that are declassed to levels greater than k. Therefore, after a certain number of updates, a β-tree need to be rebuilt. However, this work can be done off-line at predefined times when, for instance, the size of the β-tree decreases under a certain threshold.

The result of the first experiments have demonstrated a good performance of both the Beta algorithm and the algorithms presented in this paper for the management of β-trees in front of tuples updates. We are currently producing quantitative results and we are also carrying out a comparative study that takes into account different implementation choices.

6 Conclusions

In this paper we have consider the qualitative approach to user preferences in which a binary preference relation is defined among objects and a special operator (called Best) is used to extract relevant data according to such preference relation. For this operator, we have proposed and studied a special index structure, called β-tree, which can be used for a rapid evaluation of the Best operator. To our knowledge, this is the first proposal of index structure for preference queries. We have presented a number of practical algorithms for the efficient maintenance of β-trees in front of database updates. Although the algorithms have an inherent quadratic complexity, they perform very rapidly in the vast majority of cases. In fact, the algorithms reduce at minimum the number of operations required by operating over entire subtrees of β-trees. These results provide the basis for the development of a practical preference systems for dynamic data intensive applications.

References

1. R. Agrawal and E. L. Wimmers. A Framework for Expressing and Combining Preferences. In *Proc. of the 2000 ACM SIGMOD Int. Conference on Management of Data, USA*, pp. 297–306, 2000.
2. S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *Proc. 17th Intl. Conf. on Data Engineering, Heidelberg, Germany*, pp. 421–430, 2001.
3. J. Chomicki. Querying with Intrinsic Preferences. In *Proc. 8th International Conf. on Extending Database Technology, Czech Republic.*, 2002.
4. R. Fagin and E.L. Wimmers. Incorporating User Preferences in Multimedia Queries. In *Proc. 6th Intl. Conf. on Database Theory, Greece*, pp. 247–261, 1997.
5. V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. In *Proc. of ACM SIGMOD Int. Conference on Management of Data, USA*, pp. 259–269, 2001.
6. W. Kiessling. Foundations of Preferences in Database Systems. In *Proc. of International Conference on Very Large Data Bases*, 2002.
7. R. Torlone, P. Ciaccia. Which Are My Preferred Items? In *Proc. of Workshop on Recommendation and Personalization in eCommerce, Spain*, pp. 217–225, 2002.
8. R. Torlone, P. Ciaccia. Finding the Best when it'a a Matter of Preference. In *X Convegno su Sistemi Evoluti per Basi di Dati, Italia*, pp. 347–360, 2002.
9. J. van den Bercken, B. Blohsfeld, J. P. Dittrich, J. Kramer, T. Schafer, M. Schneider, and B. Seeger. XXL – a library approach to supporting effcient implementations of advanced database queries. In *Proc. of 27th Intl. Conf. on Very Large Data Bases, Roma, Italy*, 2001.