Efficient Rewriting Algorithms for Preference Queries

Periklis Georgiadis[#1], Ioannis Kapantaidakis[#2], Vassilis Christophides[#,¤3]
Elhadji Mamadou Nguer[*4], Nicolas Spyratos[*5]

[#]*Department of Computer Science, University of Crete, Greece*
[1]perge@csd.uoc.gr
[2]jkapad@csd.uoc.gr
[¤]*Institute of Computer Science, Foundation for Research and Technology-Hellas, Greece*
[3]christop@ics.forth.gr
[*]*Laboratoire de Recherche en Informatique, Université Paris Sud, France*
[4]nguer@lri.fr
[5]spyratos@lri.fr

***Abstract*— Preference queries are crucial for various applications (e.g. digital libraries) as they allow users to discover and order data of interest in a personalized way. In this paper, we define preferences as preorders over relational attributes and their respective domains. Then, we rely on appropriate linearizations to provide a natural semantics for the block sequence answering a preference query. Moreover, we introduce two novel rewriting algorithms (called LBA and TBA) which exploit the semantics of preference expressions for constructing progressively each block of the answer. We demonstrate experimentally the scalability and performance gains of our algorithms (up to 3 orders of magnitude) for variable database and result sizes, as well as for preference expressions of variable size and structure. To the best of our knowledge, LBA and TBA are the first algorithms for evaluating efficiently arbitrary preference queries over voluminous databases.**

I. INTRODUCTION

With the Web explosion, an increasing number of users access large data collections without a precise knowledge of their content, or a clearly identified search goal. Users would rather describe features of data that are potentially useful in some task, or in other words features that best suit their preferences. Modern database systems should then be able to process queries enhanced with preferences, and such queries are called *preference queries*. The answer to a preference query is a sequence of data blocks, where each block contains data that are more interesting (in terms of the preferences) than the data in the following block. In this way, the user can inspect the blocks one by one and stop inspection at any point at which he feels satisfied by the data already inspected. In this paper, we are interested in the efficient computation of such block sequences when data collections are modeled as relational tables and preferences as binary relations over the table attributes and their respective domains.

A. Motivating Example

Consider table R(W, F, L) in Fig.1, describing part of the contents of a Digital Library (DL), where for simplicity each tuple is identified by a tuple identifier (tid). A student wishing to write an essay on European writers might state the following preferences over DL resources:

(1) Joyce is preferred to Proust or Mann (preference P_W),
(2) odt and doc format are preferred to pdf (P_F),
(3) English is preferred to French, and French to German (P_L).

He might also state that:

(4) Writer (W) is as important as Format (F), whereas the Writer-Format combination is more important than Language (L).

Such statements define actually binary relations, called *preference relations*: relations (1), (2) and (3) are defined over attribute domains, whereas (4) over the set of attributes. Preference relations are usually required to satisfy some intuitive properties like reflexivity and transitivity, that is to be preorders ([5], [18], [31]). Note that a preference relation can be expressed over an attribute domain independently of whether the domain is naturally ordered (e.g. timestamp of a DL resource) or not (e.g. format of a DL resource).

Let us consider first the preference P_W which relates three values of the attribute W, namely *Joyce*, *Proust* and *Mann*. The underlying assumption here is that the only tuples that are of interest to the user are those containing one of these values. Therefore, the set of tuples matching P_W is the answer to the query Q_W: (W=*Joyce*)∨(W=*Mann*)∨(W=*Proust*). Referring to Fig. 1, the answer to Q_W is the following:

$$Ans(Q_W)=\{t1, t2, t3, t4, t5, t7, t8, t9, t10\}.$$

Now, the preference P_W partitions the set $Ans(Q_W)$ into two subsets. Indeed, as Joyce is preferred to either Proust or Mann, the subset $\{t1, t5, t7, t9\}$ with resources on Joyce is preferred to the subset $\{t4, t8, t10\} \cup \{t2, t3\}$ with resources on Mann or Proust. Therefore if PQ_W denotes the preference P_W together with the query Q_W, we feel justified in defining the *answer* to PQ_W as the following sequence:

$$Ans(PQ_W)=\{t1, t5, t7, t9\} \leftarrow \{t4, t8, t10\} \cup \{t2, t3\}$$

Suppose next that we consider the combination of two preferences, say P_W and P_F. Let's call this combined preference P_{WF}. Reasoning similarly as before, a tuple is of interest to the user only if it contains a value of W appearing

tid	W	F	L
t1	joyce	odt	en
t2	proust	pdf	fr
t3	proust	odt	de
t4	mann	pdf	de
t5	joyce	odt	fr
t6	wolf	doc	en
t7	joyce	doc	en
t8	mann	ppt	gr
t9	joyce	doc	fr
t10	mann	odt	de

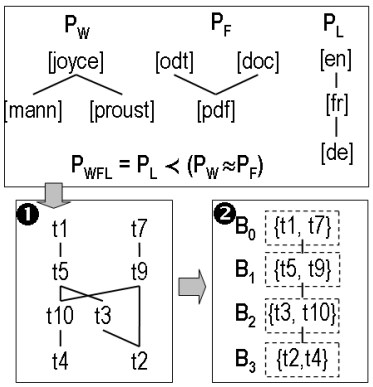

Fig 1 Preferences on a DL Relation

in P_W and a value of F appearing in P_F. Therefore, the set of tuples matching P_{WF} is the answer to the following query:

Q_{WF}: (W=*Joyce*∧F=*odt*)∨(W=*Joyce*∧F=*doc*)
∨ (W=*Joyce*∧F=*pdf*)∨(W=*Mann*∧F=*odt*)∨(W=*Mann*∧F=*doc*)
∨ (W=*Proust*∧F=*odt*)∨(W=*Proust*∧F=*doc*)
∨ (W=*Mann*∧F=*pdf*)∨(W=*Proust*∧F= *pdf*)

Referring to Fig.1, the answer to Q_{WF} is as follows:

$$Ans(Q_{WF})= \{t1, t5, t7, t9, t10, t3, t4, t2\}$$

Now, the preference P_{WF} partitions the set $Ans(Q_{WF})$ into several subsets (or blocks). However, this time, in order to find the blocks and their sequencing, we must first derive the preference P_{WF} from preferences P_W and P_F; and to do this we must consider their relative importance as expressed in statement 4 above.

As P_W is as important as P_F ($P_W \approx P_F$), given that *Joyce-odt* (*t1*, *t5*) or *Joyce-doc* resources (*t7*, *t9*) are the most preferred ones (top block), while *Mann-pdf* (*t4*) or *Proust-pdf* resources (*t2*) are the least preferred (bottom block), we obtain the following sequence of blocks as the answer to the preference query PQ_{WF} (blocks that "tie" preference-wise will be merged):

$$Ans(PQ_{WF})=\{t1, t5\}\cup\{t7, t9\}\leftarrow\{t3\}\cup\{t10\}\leftarrow\{t4\}\cup\{t2\}$$

We can easily observe that with the exception of *Proust-odt* (*t3*) and *Mann-odt* (*t10*), all other conjunctions of preference terms used to compute the middle block of $Ans(PQ_{WF})$ yield empty results. In a similar way, we can compute the block sequence illustrated in Fig.1.2 answering the preference query PQ_{WFL}. It is worth noticing that the resulting block sequence essentially linearizes the order of tuples induced by the preference P_{WFL} as depicted in Fig.1.1.

Preferences in our setting, are explicitly stated by the user either *online* (short standing preferences) or when a user first subscribes to the system (long standing preferences) [19]. Alternatively, the user may wish to obtain in the result only the *top-k* tuples (or blocks) that best suit his preferences. If so, then search terminates when k is reached (by also considering *ties*). In this paper, the main questions we address are:

(1) How can the system ensure that blocks not already computed contain only less interesting tuples preference-wise? For instance, why does block B_2 of Fig.1.2 contain tuples *t3* and *t10* but not *t2,* although *t9* is more preferable to both *t10* and *t2*?

(2) How can a preference query be rewritten to avoid costly computation of the tuple order depicted in Fig.1.1? Can we exploit user preference semantics to derive queries directly constructing the blocks of the result? (e.g. the queries W=*Joyce*∧ F=*odt* and W=*Joyce*∧F=*doc* precede W=*Proust* ∧F=*odt*)

(3) How can a preference query processor progressively exploit in various settings (*online* vs. *subscribed* preferences, *interactive* vs. *top-k* block construction) the Cartesian Product of the attribute terms (usually smaller than the number of database tuples) appearing in a user preference?

B. Contributions

In response to the previous questions, our contributions can be summarized as follows:

(a) In Section II we rely on preorders ([5], [14], [31]) to capture preferences over attributes and their domains. By *linearizing* preorder domains we can then naturally *induce* block sequences of a preference query result.

(b) In Section III we propose two novel query-rewriting algorithms, named LBA (Lattice Based Algorithm) and TBA (Threshold Based Algorithm) supporting a progressive evaluation of block sequences. Unlike the quadratic cost of existing algorithms ([6], [11], [12], [22], [33]), LBA avoids any tuple dominance testing, and accesses only those tuples (and only once) that belong to the blocks of the result. Its cost is determined by the number of the queries (eventually empty) required to construct the blocks. As this number may grow large (e.g. in long standing preferences), TBA employs appropriate threshold values to terminate tuple fetching, while dominance is tested only among each block's retrieved tuples. (c) In Section IV we experimentally evaluate our algorithms w.r.t. the database and requested result size, as well as the preference size (i.e. the number of attributes and their involved values) and structure (equally or more important attributes). In a typical scenario, requesting the top block from a 1 GB database w.r.t. a long standing preference over 5 attributes with 12 values each, LBA scales linearly and outperforms by 3 orders of magnitude dominance-testing based algorithms like BNL [6] and Best [33]. Although TBA scales quadratically, it exhibits better performance (up to 1 order of magnitude) than BNL and Best since it needs to compare a smaller fraction of the database. TBA overtakes LBA when more than 5 attributes with 12 values each are used. Still, both outperform BNL and Best up to 1 order of magnitude (especially for short standing preferences). Last but not least, the time required by BNL and Best to compute the top block suffices for computing half (one third, respectively) of the entire block sequence by LBA (TBA, respectively), as the latter two rely solely on the number of necessary queries and avoid database rescans.

Finally, in Section V we position our algorithms w.r.t. related work, while in Section VI we discuss several future extensions of LBA and TBA.

II. Modelling User Preferences

In this paper we rely on partial preorders ([5], [18], [31]) to model a preference relation. We write $d \precsim d'$ to denote that d' is *at least as preferable* as d on a domain D. Thus, the *symmetric*

part of $\precsim$ is essentially an equivalence relation modelling the *equal preference* relation $\approx$ (when both $d \precsim d'$ and $d' \precsim d$ hold), and the *asymmetric* part of $\precsim$ is a strict partial order capturing strict preference $d \prec d'$ (when $d \precsim d'$ and $\neg d' \precsim d$). As $\precsim$ is partial, an *incomparability* relation $\|$ is induced on D. In a nutshell, given a preference relation $\precsim$ over D, for any two elements d and d′ of D one of the followings may hold; $d \prec d'$, $d' \prec d$, $d \approx d'$, $d \| d'$. It should be stressed that we explicitly distinguish between equally preferred and incomparable elements, usually captured jointly in strict-order frameworks ([12], [22]) as *indifferent* elements. This explicit distinction enables to elicit user preferences in a less ambiguous way, as well as to overcome various semantic issues arising in preference composition. Moreover, our choice to rely solely on partial preorders, without any further assumptions, is driven by the fact that preference incompleteness may be uniquely or multiply resolvable, or even irresolvable [18]. Similarly, we avoid making unnecessary assumptions, and rely exclusively on the given input, by interpreting as interesting only those items that the user has referred to. Thus, as in [31], we distinguish between *active elements*, i.e. those explicitly appearing in $\prec$ or $\approx$, from *inactive* elements, with the former representing elements of interest to the user.

Combined independent preferences stated over *several* attributes as well as their relative importance are captured by preference expressions. Let $R(\mathcal{A})$ be a relational schema, where $\mathcal{A} = \{A1, .., An\}$ is a set of attributes with associated domains, and let A be a nonempty subset of $\mathcal{A}$. A *preference expression* over A, denoted P_A, is defined as follows: $P_A ::= P_{Ai} | (P_X \approx P_Y) | (P_X \prec P_Y)$, where $X \cup Y = A$ and $X \cap Y = \emptyset$. P_{Ai} denotes a preference relation over A_i, while $\approx$ is an equivalence relation and $\prec$ a strict ordering relation over A, extending the Pareto and Prioritization composition semantics to our model.

Definition 1: Given a preference expression $P_X \approx P_Y$, we define an induced relation $\precsim_{PXY}$ in dom(X)×dom(Y), as:

$(x, y) \prec_{XY} (x', y')$ iff $(x \prec_X x' \wedge y \precsim_Y y') \vee (x \precsim_X x' \wedge y \prec_Y y')$

$(x, y) \approx_{XY} (x', y')$ iff $x \approx_X x' \wedge y \approx_Y y'$

$(x, y) \|_{XY} (x', y')$ otherwise.

Definition 2: Given a preference expression $P_Y \prec P_X$, we define an induced relation $\precsim_{PXY}$ in dom(X)×dom(Y), as:

$(x, y) \prec_{XY} (x', y')$ iff $x \prec_X x' \vee (x \approx_X x' \wedge y \prec_Y y')$

$(x, y) \approx_{XY} (x', y')$ iff $x \approx_X x' \wedge y \approx_Y y'$

$(x, y) \|_{XY} (x', y')$ otherwise.

The third case in each of the above definitions although redundant, has been included to maintain our distinction between equally preferred elements and incomparable ones. Thus, Def.1 differs from frameworks which do not distinguish preference incomparability as a separate case in the absence of strict preference ([11], [21], [26], [28]), while both Def.1 and Def.2 differ from the respective ones of [12] and [22]. Those in [12] cannot preserve a strict partial order composition result, while the ones in [22] fail to retain associativity. The former is shown in [11]; for the latter, consider tuples (x1, y1, z1), (x1, y1, z2) with $z1 \prec_Z z2$. Suppose we first apply Prioritization [2] or Pareto on X and Y (the leftmost two attributes); the result would be $(x1, y1) \|_{XY} (x1, y1)$. If we went on to compose this intermediate result with Z, the final result would be $(x1, y1, z1) \|_{XYZ} (x1, y1, z2)$, instead of $(x1, y1, z1) \prec_{XYZ} (x1, y1, z2)$; *q.e.d.* Associativity of both compositions and closure of preorders under them (both achieved with Def.1 and 2) enable a bottom-up evaluation of arbitrary preference expressions.

A preference query PQ over a relation R is defined by a preference expression P_A, together with an optional integer k limiting the required result size. Based on P_A one can induce (a) a preorder $\precsim_{PA}$ over the corresponding Cartesian Product of attributes [32], and (b) a final preference relation $\precsim_T$ over R through projection over A.

A partial preorder is a construct that users may find difficult to grasp (see Fig.1.1). Instead, we rely on block sequences, i.e. ordered partitions [31]. In such a sequence, each block contains preference-wise incomparable elements; the top block contains the most preferred elements, and in every other block, for each element, there exists a more preferred element in the preceding block. This relation, that we call a *cover relation*, is a powerdomain set order, similar to the subtyping [8] and Hoare relation [7], proven to be a partial order when derived from a preorder relation, and thus a total order for a partition. A block sequence is computed by iteratively extracting the next maximal element[1] (i.e. a variant of topological sorting).

As we will see in the sequel, our algorithms aim to compute the block sequence answering a preference query without actually needing to construct the induced ordering of tuples. This is achieved by exploiting the semantics of a preference expression and, in particular, by linearizing the Cartesian Product of all attribute terms appearing in the expression (see Fig.2). Going one step further, we don't even need to construct and linearize this Cartesian Product. Instead, we can simply generate its block sequence from the block sequences of its constituent preference relations. In Fig.2.1 the block sequences of P_W and P_F are $\{Joyce\} \leftarrow \{Proust, Mann\}$ and $\{odt, doc\} \leftarrow \{pdf\}$ respectively. Thus, the following theorems provide the means to compute the block sequences of arbitrary preferences progressively.

Theorem 1: Given the block sequences $X_0 \leftarrow X_1 \leftarrow \ldots \leftarrow X_{n-2} \leftarrow X_{n-1}$, and $Y_0 \leftarrow Y_1 \leftarrow \ldots \leftarrow Y_{m-2} \leftarrow Y_{m-1}$ of two preferences P_X and P_Y, the block sequence $Z_0 \leftarrow Z_1 \leftarrow \ldots \leftarrow Z_{n+m-3} \leftarrow Z_{n+m-2}$ of the preference induced by $P_X \approx P_Y$ over X×Y, will consist of n+m-1 blocks; each block Z_p will comprise elements only from blocks X_q and Y_r, such that q+r=p.

Theorem 2: Given the block sequences $X_0 \leftarrow X_1 \leftarrow \ldots \leftarrow X_{n-2} \leftarrow X_{n-1}$, and $Y_0 \leftarrow Y_1 \leftarrow \ldots \leftarrow Y_{m-2} \leftarrow Y_{m-1}$ of two preferences P_X and P_Y, the block sequence $Z_0 \leftarrow Z_1 \leftarrow \ldots \leftarrow Z_{n \cdot m-2} \leftarrow Z_{n \cdot m-1}$ of the preference induced by $P_Y \prec P_X$ over X×Y, will consist of n*m blocks; each block Z_p will comprise elements only from blocks X_q and Y_r, and it will hold p=q*m+r. For every value of q ranging from 0 to n-1, r will range from 0 to m-1; i.e. Z_p's will derive from $X_0Y_0, X_0Y_1, \ldots, X_0Y_{m-1}, X_1Y_0, \ldots, X_{n-1}Y_{m-1}$.

For instance, given the two block sequences $M_0 \leftarrow M_1$ and $F_0 \leftarrow F_1$ of Fig.2.1 for P_W and P_F respectively, the block

[1] Unless otherwise specified, we shall refer to classes of equivalence, of a preorder's symmetric part, rather than to single elements.

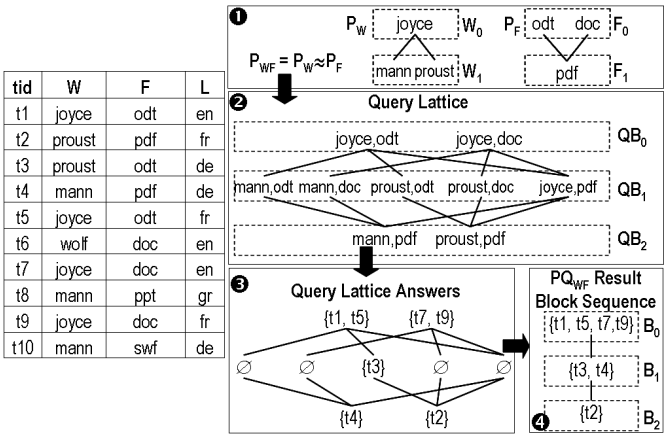

Fig. 2 Query Ordering Framework

sequence for the preference relation induced by $P_W \approx P_F$ will comprise 3 blocks (2+2-1). As seen in Fig.2.2 the top block (QB_0) will combine elements from blocks whose index sum is 0, i.e. W_0 with F_0, the second (QB_1), from blocks whose index sum is 1, i.e. W_0 with F_1, and W_1 with F_0 , and the third (QB_2), from blocks whose index sum is 2, i.e. W_1 with F_1.

III. Query-Ordering Algorithms

First we introduce the basic notation employed in the rest of this paper. By $V(P, A_i)$ we denote the set of *active terms* for preference P_{Ai}, over attribute A_i, i.e. $V(P,A_i) \subseteq dom(A_i)$ (e.g. V(P,W)={*Joyce, Proust, Mann*}, in Fig.2). Given a preference P_A over a non-empty subset A of R's attributes, dom(A) is used to denote the Cartesian Product $\times_i(dom(A_i))$, while V(P, A) the corresponding *active preference domain*; thus, V(P,A) $\subseteq$ dom(A). V(P,A) essentially represents the product of active attribute terms, regardless of whether they are actually instantiated. Moreover, T(P,A) is used to denote the set of *active tuples* of R featuring active terms for every attribute of P_A (all other tuples are called *inactive*); it holds that $\pi_A(T(P,A)) \subseteq V(P,A)$. For example, in Fig.2, $T(P_{WF},\{W,F\})=\{t1, t2, t3, t4, t5, t7, t9\}$ (note that w.r.t Fig.1 we changed the value of attribute F in tuple *t10* from *doc* to *swf*). The *preference density* d_P of a preference expression P_A is defined as |T(P,A)|/|V(P,A)|, whereas its *active ratio* a_P as |T(P,A)|/|R| (e.g. $d_{P_{WF}}$=7/9 and $a_{P_{WF}}$=7/10, in Fig.2).

A. The Query Lattice

An expression P_A over a set of attributes $A=\{A_{i1}, A_{i2}, ..., A_{iN}\}$ induces a preference over the elements (a_{k1}, a_{k2}, ..., a_{kN}) of the active preference domain V(P,A). These elements are essentially conjunctive queries of the form $A_{i1}=a_{k1} \wedge A_{i2}=a_{k2} \wedge ... \wedge A_{iN}=a_{kN}$, which when executed will retrieve the active tuples T(P,A). Henceforth, with a slight abuse of terminology, we shall call the respective ordering of queries the *Query Lattice*.

Consider, for example, the preference expression $P_{WF}=(P_W \approx P_F)$, $P_W=\{Proust \prec Joyce, Mann \prec Joyce\}$, $P_F=\{pdf \prec odt, pdf \prec doc\}$ of Fig.2.1. Fig.2.2 shows the induced preference P_{WF} over the Cartesian Product of the two active domains V(P,W) and V(P,F); it also depicts the *induced* block sequence on $V(P_{WF},\{W,F\})$. Then, to compute the top block B_0 of the preference query PQ_{WF} we need to execute the queries W=*Joyce*$\wedge$F=*odt* and W=*Joyce*$\wedge$ F=*doc* deriving from the first query block QB_0. As both queries have non-empty results ({*t1*, *t5*} and {*t7*, *t9*}, respectively, see Fig.2.3), they will return the *only* maximal tuples of our relation, as the top block B_0 of $T(P_W \approx P_F,\{W,F\})$ (see Fig.2.4).

However, not every query in the lattice is guaranteed to be non-empty. Consider, for instance, that the user is interested in obtaining the next block of $T(P_W \approx P_F,\{W,F\})$. As seen in Fig.2.3, from the five queries of the second lattice block QB_1, only W= *Proust*$\wedge$F=*odt* has a non-empty result ({*t3*}) which belongs to the next block of maximals B_1 in $T(P_W \approx P_F,\{W,F\}) \backslash B_0$. Yet, all other maximals, if any, have to result from queries that are successors (recursively, their successors, in case they are empty) of the empty queries in QB_1, and at the same time, are not successors of any other non-empty query in QB_1. This is the case of W=*Mann*$\wedge$F=*pdf* in QB_2 (with result {t4}) being child of the empty query W=*Mann*$\wedge$F=*odt* and, at the same time, unrelated to the non-empty query W=*Proust* $\wedge$F=*odt* of QB_1. On the contrary, W=*Proust*$\wedge$F=*pdf* in QB_2, although it is a child of two empty queries in QB_1, it is also a child of the following non-empty query: W=*Proust*$\wedge$F=*odt* of QB_1; thus, its answer is not a maximal, and so it does not qualify for B_1. Recursively, we can compute the bottom block B_2 of PQ_{WF} as shown in Fig.2.4.

B. Lattice Based Algorithm (LBA)

Algorithm LBA takes as input a relation R and a preference expression P_A involving a subset A of R's attributes. Then, it outputs progressively successive blocks of T(P,A). Each time a block is computed, the user may signal to continue with the next one; alternatively, he may request to obtain the top-k tuples of T(P,A).

Algorithm LBA
input: a relation R, a preference expression P_A and a k>0
output: the block sequence of T(P, A)

```
1:  QB = ConstructQueryBlocks(P_A.root)
2:  totalsize = i = 0
3:  repeat
4:    Uq_i = GetBlockQueries(QB[i])
5:    totalsize += Evaluate(Uq_i)
6:    i += 1
7:  until ExitReq or totalsize >= k or i=|QB|
```

To this end, LBA relies on an internal representation of the sequence of blocks of an active preference domain V(P,A) (see Fig.2.2). In particular, an array QB is used to hold in main memory only the structure of the block sequence of V(P, A). The corresponding Query Lattice is not materialized but rather the queries needed to generate the blocks B_i of T(P,A) are computed and executed on the fly. Each QB entry is essentially a list whose elements hold only the block indices of the active terms of $V(P,A_i)$ forming a block of V(P,A). Going back to Fig.2, QB_0 contains the singleton list <0, 0>, for W_0, F_0, whereas QB_1 contains the list <0, 1>←<1, 0>, for W_0, F_1 and W_1, F_0, respectively. After computing QB (line 1), LBA iteratively calls *GetBlockQueries* (line 4) to create the associated list of conjunctive queries and *Evaluate* (line 5) to

Function *ConstructQueryBlocks*
input: a preference expression P_A
output: a query block sequence QB

```
1:  if P is a leaf then          //a preference relation P on attribute A_i
2:    QB = PrefBlocks(V(P,A_i))
3:  else
4:    QB_left = ConstructQueryBlocks(P.left)
5:    QB_right = ConstructQueryBlocks(P.right)
6:    if P.type = '≈' then       // equally important preferences
7:          for w=0 to |QB_left| + |QB_right| - 1
            //construct the block sequence of V(P.left ≈ P.right, A)
8:        QB[w] = ∪{QB_left[i] × QB_right[j] | i+j=w}
9:    else                       // strictly more important preferences
10:     w=0
11:     for j=0 to |QB_right|-1
12:       for i=0 to |QB_left|-1
            // construct the block sequence of V(P.left≺P.right, A)
13:       QB[w] = QB_left[i] × QB_right[j]
14:       w+=1
15: return QB
```

Function *Evaluate*
input: a list of queries Uq_i
output: the next block B_i

```
1:  for each q_i in Uq_i
2:   if q_i not in SQ then
3:       if ans(q_i) != ∅ then
4:           CurSQ ∪= {q_i}; B_i ∪= ans(q_i)
5:       else FQ ∪= {q_i}
6:   else FQ ∪= {q_i}
7:  while FQ != ∅
8:   for each q in FQ
9:       FQ \= {q}
10:      Q = {q | q=child(q)}
11:  for each q in Q
12:      if q not in SQ then
13:          if not q in succ(q′) forall q′ in CurSQ then
14:              if ans(q) != ∅ then
15:                  CurSQ ∪= {q}; B_i ∪= ans(q)
16:              else FQ ∪= {q}
17:      else FQ ∪= {q}
18: SQ∪ = CurSQ; CurSQ = ∅
19: print B_i, return |B_i|
```

output successive T(P, A) blocks (keeping track, so non-empty queries are executed only once) until termination is signaled (*ExitReq*) or V(P, A) is exhausted.

ConstructQueryBlocks traverses recursively a preference expression tree P_A (from P.root) and computes bottom-up the number of blocks and their origin in QB. For each QB entry it generates the structure of the respective block sequence when ≈ (lines 7-8) and ≺ (lines 10-14) appear as a preference relation between expressions P.left and P.right (Theorems 1 and 2). For leaves (i.e. the preference relations over the individual attribute domains $V(P,A_i)$), the respective QB entries are computed (line 2) by *PrefBlocks*. For example, in its "bottom left" recursion step *ConstructQueryBlocks* creates a QB with two entries for the block sequence $W_0 \leftarrow W_1$ of P_W.

Evaluate executes each query q_i of its input set Uq_i. It keeps track of non-empty queries in SQ, so that they are executed only once. Also, for the tuple block of T(P,A) currently processed, it keeps track of non-empty queries in CurSQ (line 4) and of empty ones in FQ (line 5). For each non-empty query it appends its answer to current block B_i. For empty ones, it applies (lines 11 to 17) the previous process on their immediate (or transitive) successors which are not in SQ (i.e., avoiding to execute twice a non-empty query), and not in CurSQ (i.e., ensuring they are not at the same time successors of any non-empty query). This process is terminated when no more successors are available (line 11) or there are no more empty queries to inspect (line 17). Finally, *Evaluate* outputs the computed block and returns its size (line 19).

C. The Threshold Values

When |V(P,A)|>>|T(P,A)|, LBA will be forced to execute several queries which may yield empty results. For this reason, we devise a second algorithm, called TBA, which is a hybrid of the Query Lattice presented previously and the algorithms performing dominance tests ([6], [33]). However, in order to compare as few database tuples as possible, TBA relies on threshold values of the active preference domain V(P,A).

Let us return, for example, to the preference expression $P_{WF}= (P_W \approx P_F)$ of Fig.2. The top block QB_0 (see Fig.2.2) of the induced Query Lattice $V(P_W \approx P_F, \{W,F\})$ contains the maximal values of the active preference domain, since it combines elements from the top blocks W_0 and F_0 of V(P,W) and V(P,F). It is obvious that the corresponding value pairs on W or F behave as *thresholds*. For instance, there cannot be any tuple not inspected yet in the result of PQ_{WF}, that has better values than <*Joyce*, *odt*> and <*Joyce*, *doc*>. Let us now consider, a disjunctive query q on attribute W formed by all active terms of W_0; in our example, q is W=*Joyce* as there is only one value in W_0. Clearly, any tuple of R, not belonging to the result of q, cannot be better than tuples matching pairs of active terms obtained by the next block W_1 of V(P,W), i.e. the pairs $W_1 \times F_0$={<*Mann, odt*>, <*Mann, doc*>, <*Proust, odt*>, <*Proust, doc*>}. In other words, we lower the threshold by going one block "down" in V(P,W) (i.e. the active terms of the attribute we chose to execute q), while we keep the previous threshold for V(P,F). Then, we need to check for dominance among the tuples returned by q. In our example, we derive *t1*≈*t5*, *t7*≈*t9*, and *t1*||*t7*, and thus, all tuples are undominated. Due to transitivity, if the new threshold $W_1 \times F_0$ is covered by the set of undominated tuples of ans(q), the latter actually constitutes B_0, i.e. the undominated tuples of T(P,A). Repeating the above process, we reach the final block B_2 of $V(P_W \approx P_F, \{W,F\})$ and construct the block sequence of tuples as depicted in Fig.2.4.

D. Threshold Based Algorithm (TBA)

TBA calls *PrefBlocks* (line 2) to compute the block sequences of active attribute domains $V(P,A_i)$. Similarly to LBA, it maintains the result in an array PB of lists whose elements hold only the block indices of the active terms of $V(P,A_i)$. The threshold values are stored in an array *Thres* of size m (i.e. the total number of attributes A_i), and initially comprise the top blocks of all PB lists (line 3). Throughout its execution, TBA keeps in memory two sets with the tuples that were fetched, but not yet returned: *(D)ominated* contains the tuples for which some better one was found and

Algorithm TBA
input: a relation R, a preference expression P_A and a k>0
output: the block sequence of T(P,A)

```
1:  for j=1 to m                    // m is the number of attributes in P_A
2:    PB[j] = PrefBlocks(V(P, Aj))
3:    Thres[j]=head(PB[j])
4:  U=D=∅; Totalsize=0
5:  repeat
6:    i = min_selectivity(Thres)
7:    Q = ∨(A_i=v_j), ∀v_j∈Thres[i]
8:    <D, U> = OrderTuples(ans(Q), D, U)
9:    if next(PB[i]) then
10:     Thres[i] = next(PB[i])
11:     <D, U, Totalsize >= CheckCover(U, D)
12:   else
13:     Thres={⊥}
14:     <D, U, Totalsize >= CheckCover(U, D)
15:     exit
16: until Totalsize >=k or ExitReq
```

Function *OrderTuples*
input : sets of tuples A, Dom, set of tuple classes Und
output: a pair of sets <UptDom, UptUnd>,
UptUnd: set of tuple classes, UptDom: set of tuples

```
1:  UptDom=Dom                      // [] denotes a class of tuples
2:  if Und=∅ then UptUnd = {[t1]} else UptUnd=Und
                                    // t1 is the first active tuple of A
3:  for each active tuple t in A
4:    IsDominated=false
5:    for each t′ in UptUnd
6:      if t<t′ then
7:        IsDominated =true
8:        UptDom ∪={t}; break   //inner for
9:      elseif t~t′ then
10:       [t′] ∪= {t}; break   //inner for
11:     elseif t′<t then
12:       UptUnd \= {[t′]}
13:       UptDom ∪= {t′}
14:   if not IsDomilnated then
15:     UptUnd ∪= [t]
16: return <UptDom, UptUnd>
```

(U)ndominated the equivalence classes of tuples for which a better tuple is yet to be met. Then, the following four steps are repeated until the requested answer size is reached or the user signals termination (line 16), or T(P,A) is exhausted (line 12):

(a) TBA identifies the lowest selectivity $V(P,A_i)$ block (for all its active terms), among those referred by *Thres* (line 6) and executes the respective disjunctive query;

(b) *OrderTuples* is called (line 8) to pair-wise compare returned tuples and update accordingly sets *D* and *U*;

(c) the next best $V(P,A_i)$ block updates *Thres* (line 10);

(d) *CheckCover* is called (line 11) next to output one or more blocks of the answer, and update sets *D* and *U*.

Let us turn our attention to the TBA termination condition where T(P,A) is exhausted before the user signals to exit or k is reached. This occurs when one of the list elements of *Thres* is exhausted (line 12). We prove this condition by reduction to the absurd. Assume that the block sequence of $V(P,A_k)$ is exhausted and yet there is an element $v'=<a_1, \ldots, a_{k-1}, a_k, a_{k+1}\ldots>$ of V(P,A) not already processed; v′ should contain active terms on every attribute. Thus, a_k term is either active and should have already been inspected, or belongs to the remaining part of $V(P,A_k)$. Both cases contradict the hypothesis. This termination condition is treated through a special bottom threshold, denoted as {⊥} (line 13). Then, *CheckCover* (line 14) will find any set of undominated tuples better than {⊥}, and thus will output the next blocks as requested. Function *OrderTuples* takes as input two sets of tuples, A and *Dom*, as well as a set of equivalence classes of tuples *Und*. If empty, *Und* is initially filled with the class of the first tuple of A (line 2).

OrderTuples updates the sets *Dom* and *Und* after comparing every tuple *t* of A against a single representative *t′* of all tuple classes in *Und*. Four cases may occur: (i) If *t* is found worse than some *t′* (line 6), it is appended to *Dom* and it does not have to be compared against the rest of *Und*; (ii) If *t* is found equally preferred to some *t′* (line 9), it is appended to the class of *t′* in *Und* and again no more comparisons against the rest of *Und* are needed; (iii) If *t* is found better than some *t′* (line 11), the (flattened) class of *t′* is moved from *Und* to *Dom*; *OrderTuples* continues testing *t* with the rest of *Und*; (iv) If *t* is incomparable to *t′*, comparisons continue with the rest of *Und*, without any further action. At the end of comparisons, if *t* is found not to be dominated by any *Und* element (line 14), a new class containing *t* is appended to *Und*. It should be stressed that algorithms like BNL [8] and Best [33] rely on a similar function to order tuples.

CheckCover takes as input a set of dominated tuples (*Dom*), a set of undominated tuple classes (*Und*) and the current value of the returned answer size (*CS*). Using the current threshold (*Thres*), the required *k*, and its input parameters, it recursively outputs as many blocks of the answer as possible. When finished, it returns updated versions of its input parameters. *CheckCover* checks whether *Und* covers the threshold (line 2). If so, *Und* is the next answer block B_i, and then the current answer size is updated while the set of undominated tuple classes is reset (lines 3-4). If more blocks are requested, or *k* is not reached (line 5), *OrderTuples* is employed to partition the tuples of *UptDom* in undominated and dominated ones (lines 6-7). With the sets updated in the previous step, *CheckCover* will be recursively applied (line 8), until either of the conditions in lines 2 or 5 fail.

E. Analytical Evaluation

In this section we analyze the complexity of our algorithms by focusing on the cost of computing the top block of a preference query result. As a matter of fact, generating the top block has the same cost in the worst case as constructing the entire block sequence.

The cost of LBA is mainly due to the number of conjunctive queries it has to execute in order to construct a block of the answer. A conjunctive query is usually evaluated by traversing the available indices on the involved attributes, intersecting the tids and then fetching the matching tuples from the disk. When (unclustered) B+-trees are used (to also support range queries as part of our future work) the I/O cost for each such query q will be O(log|R|+|ans(q)|). Assuming that r queries are executed in total to construct the resulting

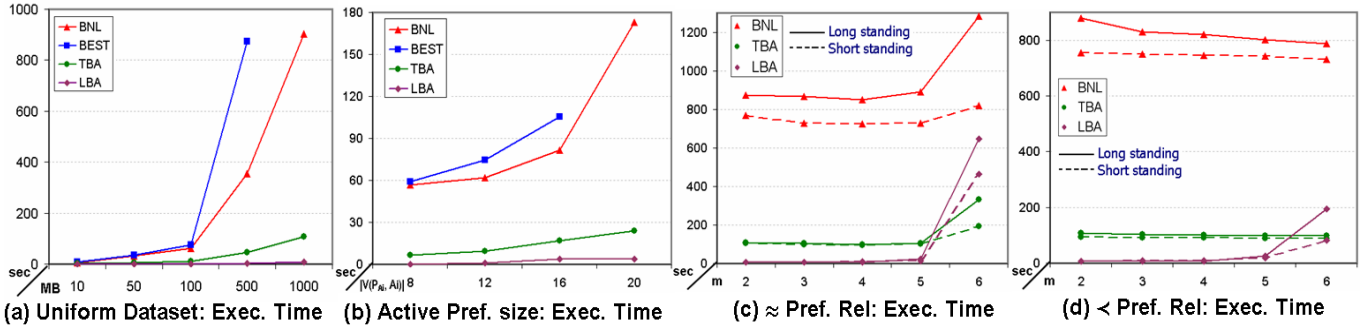

Fig 3 Execution Times for various experimental setups

block sequence, the LBA cost is O(r*(log|R|+ |ans(q)|)). In the best case, only one query is required to construct B_0 and the number of returned tuples is very small (especially for uniform data distributions). In particular, when $|V(P,A)|<<|T(P, A)|$, i.e. the preference density $d_P>>1$, the practical cost of LBA drops to O(log|R|). In the worst case, all the lattice queries need to be executed to construct the entire block sequence (i.e. *k* is omitted) as just a few of the leaf queries actually return almost all of the active tuples (especially for skewed data distributions). Thus, the total cost of the index traversals will rise to O(|V(P,A)|*log|R|) where $|V(P,A)|= \times_i|V(P,A_i)|$, while the I/O cost of their non-empty answers will be O(|T(P,A)|), bringing the total worst case cost up to $O(\times_i|V(P,A_i)|*\log|R|+|T(P,A)|)$. In particular, when $|V(P,A)|>>|T(P,A)|$, i.e. $d_P<<1$, the practical complexity of LBA in the worst case becomes O(|V(P,A)|).

Turning to TBA, the cost per executed query remains unchanged; however, queries involve now only disjunctions of preference terms per attribute, while the returned tuples are not exclusively active, but may include inactive ones matching at least one active attribute term. In addition, the fetched tuples are compared pair-wise. In the best case, one query (usually from the top lattice block) is also sufficient for constructing B_0 and the number of returned tuples is very small (i.e. we employ high selectivity queries). Thus, the cost of pair-wise dominance testing can be neglected. In particular, when the preference density $d_P>>1$ the best case practical cost of TBA is O(log|R|). In the worst case, TBA exhausts all but the last block of the query lattice, and the query executed in the next round actually returns almost all of the active tuples. The total number of queries executed in this case is given by the number of blocks of preference terms per attribute $\Sigma i|B(P,A_i)|$. An active tuple may be fetched at least once and at most m times (by m queries on different attributes), while an inactive from zero to m-1 times (depending on the number of active terms it contains and the queries on the respective attributes). Assuming a combined factor c [2] of all tuples fetched w.r.t. the number of active ones, in the worst case TBA cost is $O(\Sigma i|B(P,A_i)|* \log|R|+c*|T(P,A)|)$ for I/Os and $O(|T(P,A)|^2)$ for main memory tuple comparisons. In particular, when $|T(P,A)|>>\Sigma i|B(P,A_i)|$, the practical complexity of TBA in the worst case becomes $O(|T(P,A)|^2)$. Finally, regarding memory requirements, LBA holds a small compressed form of block sequences, while TBA holds in memory the sets *D* and *U*, at worst of size |T(P,A)|.

[2] Recall that TBA uses the most selective attribute terms, so relatively few inactive tuples are expected to be fetched.

IV. Experimental Evaluation

LBA and TBA were evaluated and compared against two widely used algorithms, namely BNL [6] and Best [33] on a P4-2.66GHz/1GB (20GB data disk) Windows XP-Pro-SP2 system, all implemented in Java on top of PostgreSQL 8.1.

Testbeds employed relations with 10 attributes with respective active domains of 20 values. Database tuples were 100 bytes long, while B+-trees indices were used. The default preference expression was $P=P_Z\prec(P_X\approx P_Y)$ while we were interested in obtaining the top block B_0. Due to its size, P is a typical example of a long standing preference. The experimental results reported in this paper ware obtained for a uniform data distribution (but *correlated* and *anti-correlated* synthetic databases ([6], [9], [27], [34]) all algorithms exhibit the same performance trends; see [20] for details).

As a common ground for performance comparison of all algorithms, we identify four major factors, namely, the *database* and requested *result size*, as well as the *preference dimensionality* and *cardinalities*. The *dimensionality* (i.e. the number of attributes involved in a preference expression) and *cardinalities* of preference expressions (i.e. their active domain sizes) are the two main parameters affecting the *size* and *structure* of V(P, A). On the other hand, keeping the rest of the factors fixed, the database size |R| affects |T(P, A)|. The relationship between |V(P, A)|, |T(P, A)| and |R| is essentially the *preference density* d_P and *active ratio* a_P. It should be stressed that, for all employed datasets, a single file scan sufficed for the retrieval of the top block by BNL and Best; this is not always the case for typical datasets, yet we followed this approach to provide a non-biased basis for the evaluation against our algorithms. Thus, all performance figures presented in the sequel for B_0, refer to a single scan for BNL and Best, which was in their favor.

The effect of database size: We scaled up the size of the database from 10 to 1,000 MB (or from 100K to 10,000K tuples). Given a preference expression P, V(P,A) is fixed, and consequently T(P,A) and density d_P increase as the database size increases, while a_P remains fixed. An alternative approach would be to fix T(P,A), and thus d_P, and consequently to decrease a_P as the database size increases. However, this setup is not useful for studying the behavior of LBA and TBA. By keeping V(P,A) and T(P,A) fixed we can't really impact either

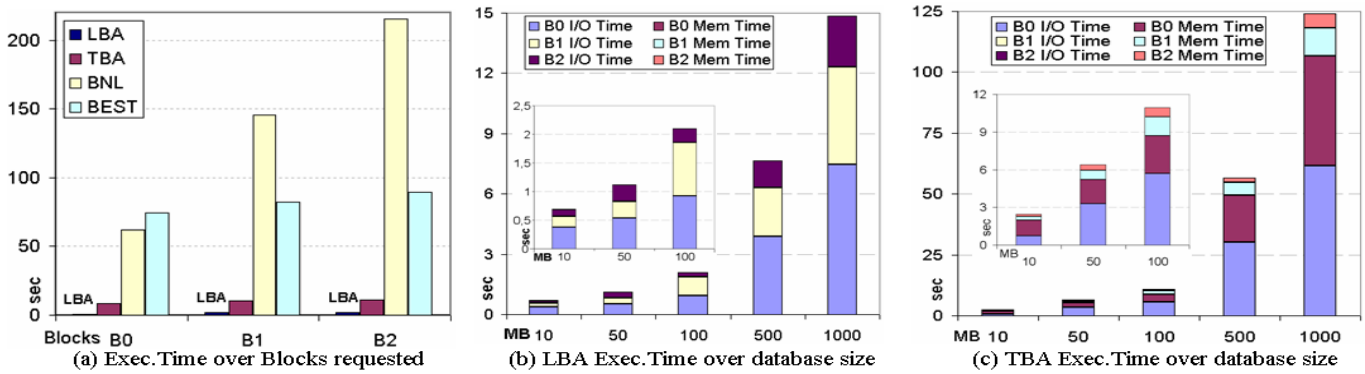

Fig 4 Scalability over blocks requested and over data size

the queries required to evaluate (yielding eventually empty results), or their matching tuples, both affecting the performance of our algorithms.

As shown in Fig.3a, LBA outperforms all others by several orders of magnitude (e.g. compare BNL over 900 sec with LBA 7 sec on a 1,000 MB database, or an improvement of almost 3 orders). For LBA, this is due to the fact that, as d_P grows well above 1, queries of the first Query Lattice block most probably suffice for computing the answer (in our testbed we need to execute only |X0|×|Y0|×|Z0|=6 queries), regardless the fact that their answer size has grown. Since TBA will not require in this case any threshold renewal, only 1 query is also executed, and thus only a small portion of the database will be finally accessed. For this reason, TBA also outperforms BNL and Best up to 1 order of magnitude, with its performance excellence rapidly increasing as the database grows. In this specific experiment, TBA fetched only 5% of the database tuples, which included almost 8% of active tuples and just 4% of the inactive ones, and thus performed only a 7%-10% fraction of the dominance tests required by BNL and Best. As more dominance tests are executed, the performance of BNL and Best fall rapidly with database growth; both proved very sensitive to the database size, and it is worth noting that, above 100 MB, Best performed poorer even than BNL. This was due to its increased memory requirements, which led to an extensive use of the Java garbage collector. Above 500MB, Best fails to terminate successfully.

The effect of preference cardinalities and dimensionality: To study the effect of the preference cardinalities we vary $|V(P,A_i)|$ for each attribute of our default expression P. In particular, we scale up $V(P,A_i)$ from 4 (representing short standing preferences) to 20 values, covering essentially the entire domains of A_i, and thus progressively increase T(P,A) up to the database size. Again, no new $V(P,A_i)$ blocks were added, to provide a common ground for our experiments. By increasing preference cardinalities, T(P,A) and a_P increase too, while density d_P remains fixed. In this setting, LBA clearly outperforms BNL and Best, this time by 2 orders of magnitude. Having to process less active tuples (8% to 12%), TBA proves to be much faster than BNL, especially, the larger each $|V(P,A_i)|$ gets. Best performs even worse and eventually crashes, running out of memory (Fig.3b).

To study the effect of preference dimensionality, we employ a 1,000 MB testbed and vary m from 2 to 6 attributes. In particular, we consider two preference expressions, an expression $P_{\approx}$ comprising only preference relations of type ≈, and an expression $P_{\prec}$ comprising only preference relations of type ≺. Clearly, as we increased the dimension m of both $P_{\prec}$ and $P_{\approx}$ on the same database, $V(P_{\prec},A)$ and $V(P_{\approx},A)$ increased too, while $T(P_{\prec},A)$ and $T(P_{\approx},A)$, respectively, decreased. Thus, the respective densities $d_{P\approx}$ and $d_{P\prec}$ decreased as well, passing from values over 1, to values below 1 (in our experiments when m changed from 5 to 6). Density affects $|B_0|$ as well; thus, with $P_{\prec}$, as m increased $|B_0|$ decreased; with $P_{\approx}$, though, $|B_0|$ decreased only for as long as $d_{P\approx}$ stayed above 1, until it started increasing again, as $d_{P\approx}$ went lower. This behavior is due to the semantics of relations ≈ and ≺. Given the imposed left-to-right order, only ≺ ensures that B_0 members for m+1 dimensions will only come from B_0 members for m dimensions, hence increasing m will constantly decrease the size of the blocks.

Fig.3c and 3d depict the total execution time of the 3 algorithms as a function of the preference dimensionality for the default long standing preference P (solid lines). In addition, we consider a typical short standing preference (dashed lines), which comprises only the top two blocks from each constituent of P. Best is not presented as it crashed for the 1,000 MB testbed. LBA performs well while $d_{P\approx}$ (or $d_{P\prec}$) is below 1. Past this point, its performance starts to drop, as it executes more and more queries with empty results, and thus a bigger portion of $V(P_{\approx},A)$ (or $V(P_{\prec},A)$, respectively) needs to be explored. Under these circumstances, TBA wins, since it executes fewer queries (e.g. for W=6, LBA evaluated 1,572 queries for $P_{\approx}$, while TBA just 5). The performance gains become more important as m increases, especially with $P_{\prec}$, whose threshold values drop more rapidly than with $P_{\approx}$. The performance of BNL and Best, on the other hand, mostly depend on $|B_0|$, and through the latter on m, as explained previously. In our experiments, as m increased, BNL and Best performances are improved since B_0 became smaller; yet in $P_{\approx}$, these performances rapidly fell for m>5, as $|B_0|$ started growing again. For short standing preferences, although the dominance tests are much fewer, LBA and TBA still hold the same performance advantages over BNL and Best.

The effect of the requested result size: In Fig.4a we report results for a 100 MB testbed, where blocks B_0 to B_2 are requested. As expected, the overall execution time for all

algorithms increases. Yet, LBA and TBA outperform BNL by 2 and 1 orders of magnitude, respectively. BNL, and Best to a smaller extent, are more sensitive to the number of requested blocks since they need an additional database scan (or part of it for Best) and process all tuples again. On the contrary, as shown in Fig.4b-4c, our algorithms primarily rely on the number of executed queries per requested block, rather than on the number and size of the blocks.

LBA memory cost (Fig.4b) is negligible compared to I/O cost, compared to TBA (Fig.4c) performing dominance tests like BNL and Best. Finally, TBA may fetch inactive tuples too, however, the result of a single query may suffice for more than one blocks (being iteratively partitioned through dominance testing).

V. Related Work

Unlike existing qualitative preference frameworks ([10]-[12], [17], [21]-[22]), in our work we rely on partial preorders to model positive independent preferences expressed both over the values of tuple attributes, as well as over the attributes themselves. In particular, we model the relation of equal, or strictly more important preference on attributes and on their domain values in a uniform manner. By using preorders, instead of strict orders [22], we distinguish between equally preferred and incomparable tuples. Hence, we overcome in a general way various semantic issues arising in preference composition (see Section II). To address these issues in a particular preference setting in which actual incomparable items are not equivalent (vs. truly equally preferred ones), [29] rely on a heavier machinery of pairs of preorders and partial orders on which Pareto and Prioritization are defined. By considering as interesting only those tuples that the user has explicitly referred to through their attribute values, we also distinguish between active and inactive tuples, whereas in [4], [10]-[12], [17], [21]-[22] the latter, being considered incomparable to the former, end up as undominated in the top block of the query result. Furthermore, unlike frameworks based on weak orders [11], [28] (i.e. preorders in which incomparable items are prohibited) or total orders with ties [14], [24], [26] (e.g. deriving from equal scores), which impose *all* tuples of one block to be strictly better than *all* tuples of the previous (or next) block, we provide an algebraic framework which is less restrictive and more natural to interpret. It is based on cover relations over the power set of a preorder domain, and we employ it for the block sequences of preference relations over individual attributes, as well as for the tuples of the result.

More precisely, existing algorithms like Block Nested Loop (BNL) [6] and Best [33] are agnostic to preference expressions, whose semantics is captured only externally by the employed dominance testing functions. For this reason, they need to access *all* tuples of a relation R at least once and perform for every R tuple at least one dominance test. Hence, they are inadequate for large databases. Moreover, as both have to read the entire relation before returning the top block, they are not suitable for a progressive result computation, as our algorithms are. For weak orders, a single-pass variation of Best is introduced in [11]: it requires that all non equal tuples of each block are incomparable to each other, while each of them dominates (and is dominated by) every tuple of the succeeding (preceding, respectively) block. This is a very restrictive semantics compared to our cover relation. Furthermore, [26] and [28] do not distinguish preference incomparability as a separate case in the absence of strict preference. The former proposes an algorithm when small lattices are combined with a (possibly infinite) total order, while the latter presents an algorithm for pruning unnecessary dominance tests. In both cases, a much faster variant of LBA is applicable which simply skips successors of every empty query constructed from the same blocks, from which a non-empty query was executed. It should be stressed that our algorithms are independent of the specific Prioritization and Pareto semantics we employ and moreover, as seen previously, their efficiency improves if the semantics deriving from strict partial or weak orders are used instead. On the other hand, our distinction between active and inactive tuples did not bias the experimental evaluation of our algorithms, as we carefully chose testbeds for which a single file scan sufficed for BNL and Best to evaluate the top block. The only hard, yet realistic, requirement we impose is the existence of indices on the preference attributes.

Probably the most thoroughly studied fragment of qualitative preference queries is that of skyline queries; they employ preferences of equal importance while each preference essentially defines a total order of attribute values. Skyline algorithms comprise two main families; the *non-index based* ones, like BNL [6], Best [33] (or their variants [11], [28]), as well as the *index based* ones, like NN [23] and BBS [27]. As expected, the latter exhibit better query performance. Yet, to do so, a different complex index over the combination of all preference attributes is required for each possible skyline query (in general, for m attributes, 2^m-1 different skyline queries need to be accommodated). In contrast to LBA and TBA, these indices can handle only totally ordered attribute domains. The skyline algorithm proposed for partially-ordered domains in [9] relies on graph encoding techniques to transform a partially ordered domain into two total orders (using interval-based labels). We believe that the linearization (originally introduced in [31]) which is based on the cover relations of preorders provides a natural semantics for evaluating arbitrary preference queries (and not only the skyline fragment), while it avoids the cost of generating and maintaining interval-based labels for graphs. The experiments reported in [9] show that the proposed algorithms do not scale well, even for small sized databases (500 and 1,000K tuples), when the majority of the involved attributes are partially ordered. For example, for 2 totally and 1 partially ordered domains a typical execution time is 50 sec, whereas, for 1 total and 2 partial orders time rises above 1,200 secs (no results are reported for more than 2 partial orders).

Last but not least, TBA bears similarities with the threshold-based evaluation of top-k queries proposed in [15]; yet, what we assume as a threshold is a set of elements of V(P,A), rather than arithmetic scores [3], [15], [16].

VI. CONCLUSIONS

Being agnostic to the preference expression, BNL and Best are very sensitive to the size of the database and of the result; LBA and TBA, on the contrary, exploit the semantics of preferences, and thus are sensitive to their size and structure, while they scale much better w.r.t. the requested result size. For voluminous databases, LBA is best for queries with short standing preferences (typically resulting to small query lattices), while TBA wins when long standing preferences (typically resulting to larger query lattices) are used instead. The main conclusions that we draw from our experiments is that LBA scales linearly (up to 3 orders of magnitude faster) compared to BNL and Best and its performance is solely affected by the number of the potentially empty queries executed when the lattice is large. On the other hand, TBA is less affected by the size of the lattice (i.e., its depth rather than its breadth) although it scales quadratically w.r.t. the database size. Yet, TBA outperforms BNL and Best (up to 1 order of magnitude) since it needs to compare a smaller fraction of the database. Notably the time required by BNL and Best to compute the top block in a typical scenario (of 1GB database with a long standing preference over 5 attributes with 12 values each) suffices for computing almost half (one third, resp.) of the entire block sequence by LBA (TBA, resp.).

In this work, we consider unconditional, positive preferences for the presence of values over discrete attribute domains from a single relational table. We are currently studying several extensions of our framework. Combining preferences through joins for evaluating preference queries over several tables can be easily accommodated as in [24]-[25]. Conditional preferences ([1], [4], [10]-[13]) can be supported by refining the Query Lattice queries with the respective condition terms, leading to finer block sequences. The same rewriting can be also employed when preference queries feature arbitrary filtering conditions and the most selective indices (preference vs. filtering attributes) should be used to evaluate them. Preferences on the absence of values, as well as negative ones ([17], [22], [24]), can be accommodated by arranging in the preorder the position either of the active attribute terms (former case), or of the attribute sets (latter case). Finally, we are interested in extending the Query Lattice with range queries in order to support more expressive preference predicates (e.g. involving arithmetic conditions) by avoiding full data scans and complex indices proposed in [30].

ACKNOWLEDGMENTS

This work is partially supported by the EU DELOS Network of Excellence in Digital Libraries (NoE-6038-507618) and by the IST IP Project KP-LAB (IP 27490).

REFERENCES

[1] R.Agrawal, R.Rantzau, and E.Terzi, Context-sensitive ranking, In *SIGMOD* 2006.

[2] H.Andreka, M.Ryan, and P.-Y.Schlobbens, Operators and Laws for Combining Preferential Relations, *Journal of Logic and Computation*, 12(1) 2002.pp.13-53.

[3] W.-T. Balke, and U. Güntzer. Multi-Objective Query Processing for Database Systems. In *VLDB* 2004.

[4] C.Boutilier, R.Brafman, H.Hoos, and D.Poole, Reasoning with conditional ceteris paribus preference statements, In *UAI*, 1999.

[5] D.Bouyssou, and P.Vincke, Introduction to topics on preference modeling, *Annals of Operations Research*, 80, 1998, pp.i--xiv.

[6] S.Börzsönyi, D.Kossman, and K.Stocker, The Skyline Operator, In *ICDE* 2001.

[7] P.Buneman, A.Jung, and A.Ohori, Using powerdomains to generalise relational databases, *Theoretical Computer Science*, 91, 1991, pp.23-55.

[8] L.Cardelli, and P.Wegner, On Understanding Types, Data Abstraction, and Polymorphism, *Computing Surveys*, 17(4), 1985, pp.471-522.

[9] C.Y.Chan, P.K.Eng, and K.L.Tan, Stratified Computation of Skylines with Partially-Ordered Domains, In *SIGMOD* 2005.

[10] J.Chomicki, Iterative Modification and Incremental Evaluation of Preference Queries, *FOIKS 2006*, Springer, LNCS 3861, 2006.

[11] J.Chomicki, Semantic Optimization of Preference Queries, *CDB 2004*, pp.133-148.

[12] J.Chomicki, Preference formulas in relational queries, *ACM Trans.Database Syst.*, 28(4), 2003, pp.427-466.

[13] C.Domshlak, and R.Brafman, CP-Nets - Reasoning and Consistency Testing, In *KR* 2002.

[14] R.Fagin, R.Kumar, M.Mahdian, D.Sivakumar, and E.Vee, Comparing and aggregating rankings with ties, In *PODS* 2004.

[15] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *PODS* 2001.

[16] U. Güntzer, W.-T. Balke, and W. Kiessling. Optimizing Multi-Feature Queries for Image Databases. In *VLDB* 2000.

[17] B.Hafenrichter, and W.Kiessling, Optimization of Relational Preference Queries, In *Australasian Database Conference* 2005.

[18] S.O.Hansson, Preference logic, *Handbook of Philosophical Logic*, Vol.4, Kluwer, 2001.

[19] Y.Ioannidis, and G.Koutrika, Personalized Systems: Models and Methods from an IR and DB Perspective.In *VLDB* 2005.

[20] I.Kapantaidakis, *Query Ordering Algorithms for Qualitatively Specified Preferences*, M.Sc.Thesis, Univ.of Crete, Greece, 2007.

[21] W.Kiessling, Preference Queries with SV-Semantics, In *COMAD* 2005

[22] W.Kiessling, Foundations of Preferences in Database Systems, In *VLDB* 2002.

[23] D.Kossmann, F.Ramsak, and S.Rost, Shooting stars in the sky: An online algorithm for skyline queries, In *VLDB* 2002.

[24] G.Koutrika, and Y.Ioannidis, Personalized Queries under a Generalized Preference Model, In *ICDE* 2005.

[25] G.Koutrika, and Y.Ioannidis, Personalization of Queries in Database Systems, In *ICDE* 2004.

[26] M.Morse, J.M.Patel. H.V.Jagadish, Efficient skyline computation over low-cardinality domains, In *VLDB* 2007.

[27] D.Papadias, Y.Tao, G.Fu, and B.Seeger, An optimal and progressive algorithm for skyline queries, In *SIGMOD* 2003.

[28] T.Preisinger, W.Kiessling, and M.Endres, The BNL^{++} Algorithm for Evaluating Pareto Preference Queries, *Multi-disciplinary Workshop on Advances in Preference Handling* 2006.

[29] K.A.Ross, On the adequacy of partial orders for preference composition, In *DBRank Workshop* 2007.

[30] K.A.Ross, P.J.Stuckey, and A.Marian, Practical Preference Relations for Large Data Sets, In *DBRank Workshop* 2007.

[31] N.Spyratos, and V.Christophides, Querying with Preferences in a Digital Library, In Dagstuhl Seminar *Federation over the Web*, No 05182, May 2005, LNAI Vol. 3847.

[32] N.Spyratos, V.Christophides, P.Georgiadis, and M.Nguer, Semantics and Pragmatics of Preference Queries in Digital Libraries, In *Int'l Workshop on Knowledge Media Science*, 2006, Meiningen, Germany (to appear in LNAI).

[33] R.Torlone, and P.Ciaccia, Which Are My Preferred Items?, In *Recommendation & Personalization in eCommerce*, 2002.

[34] Y. Yuan, X. Lin, Q. Liu, W. Wang, J.X. Yu, and Q. Zhang, Efficient Computation of the Skyline Cube, In *VLDB* 2005.