University of Crete
Computer Science Department

QUERY ORDERING BASED TOP-K ALGORITHMS FOR QUALITATIVELY SPECIFIED PREFERENCES

by

IOANNIS KAPANTAIDAKIS

Master's Thesis

Heraklion, January 2007

University of Crete
Computer Science Department

QUERY ORDERING BASED TOP-K ALGORITHMS FOR QUALITATIVELY SPECIFIED PREFERENCES

by

Ioannis Kapantaidakis

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Author:

_____
Ioannis Kapantaidakis, Computer Science Department

Supervisory
Committee:

_____
Vassilis Christophides, Associate Professor, Supervisor

_____
Georgios Georgakopoulos, Assistant Professor, Member

_____
Ioannis Tzitzikas, Assistant Professor, Member

Approved by:

_____
Trahanias Panos, Professor
Chairman of the Graduate Studies Committee

Heraklion, January 2007

QUERY ORDERING BASED TOP-K ALGORITHMS FOR QUALITATIVELY SPECIFIED PREFERENCES

Ioannis Kapantaidakis

Master Thesis

University of Crete
Computer Science Department

Abstract

Preference modelling and management has attracted considerable attention in the areas of Databases, Knowledge Bases and Information Retrieval Systems in recent years. This interest stems from the fact that a rapidly growing class of untrained lay users confront vast data collections, usually through the Internet, typically lacking a clear view of either content or structure, moreover, not even having a particular object in mind. Rather, they are attempting to discover potentially useful objects, in other words, objects that best suit their preferences. A modern information system, consequently, should enable users to quickly focus on the k best object according to their preferences. In this thesis, modelling preferences as binary relations, we introduce efficient algorithms for the evaluation of the top-k objects.

Previous related work treated preference expressions as black boxes and dealt with the idea of exhaustively applying dominance tests among database objects in order to determine the best ones, resulting in quadratic costs. On the contrary, we advocate a query ordering based approach. Our key idea is to exploit the semantics of the input preference expression itself, in terms of both the operators and the preferences involved, to define an ordering over those queries, whose evaluation is necessary for the retrieval of the top-k objects. We introduce two novel algorithms, LBA and TBA.

LBA defines an ordering over queries which are essentially conjunctions of atomic selection conditions, containing all attributes that the user preferences involve. The algorithm ensures that the way and order in which objects are fetched respect user preferences, avoiding any dominance testing, and accessing only the top-k objects,

each of them only once. From a different angle, TBA defines an order of queries which are disjunctions of atomic selection conditions over single attributes, and uses appropriate threshold values to signal object fetching termination, ensuring that all remaining objects are worse than those fetched. Dominance tests are performed only for already retrieved objects.

Analytical study and experimental evaluation show that our algorithms outperform existing ones under all problem instances.

Supervisor: Vassilis Christophides
Associate Professor

ΑΛΓΟΡΙΘΜΟΙ ΚΟΡΥΦΑΙΩΝ-Κ ΑΠΑΝΤΗΣΕΩΝ ΒΑΣΙΣΜΕΝΟΙ ΣΕ ΔΙΑΤΑΞΕΙΣ ΕΠΕΡΩΤΗΣΕΩΝ ΓΙΑ ΠΟΙΟΤΙΚΩΣ ΚΑΘΟΡΙΣΜΕΝΕΣ ΠΡΟΤΙΜΗΣΕΙΣ

Ιωάννης Καπανταϊδάκης

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Περίληψη

Τα τελευταία χρόνια, η μοντελοποίηση και η διαχείριση των προτιμήσεων έχουν προσελκύσει ιδιαίτερη προσοχή στους τομείς των Βάσεων Δεδομένων, των Βάσεων Γνώσης και των Συστημάτων Ανάκτησης Πληροφοριών. Αυτό το ενδιαφέρον πηγάζει από το γεγονός ότι ολοένα και περισσότεροι μη ειδικευμένοι κοινοί χρήστες έρχονται σε επαφή με τεράστιες συλλογές δεδομένων, συνήθως μέσω του Διαδικτύου, χωρίς, κατά κανόνα, να έχουν μια σαφή άποψη ούτε για το περιεχόμενο ούτε και για τη δομή της πληροφορίας, χωρίς καν να έχουν ένα συγκεκριμένο αντικείμενο υπ' όψει τους. Πιο πολύ προσπαθούν να ανακαλύψουν αντικείμενα που ενδεχομένως θα τους είναι χρήσιμα, αντικείμενα, με άλλα λόγια, που ταιριάζουν καλύτερα στις προτιμήσεις τους. Συνεπώς, ένα σύγχρονο πληροφοριακό σύστημα θα πρέπει να διευκολύνει τους χρήστες στο γρήγορο εντοπισμό των k βέλτιστων αντικειμένων βάσει των προτιμήσεων τους. Στην εργασία αυτή, μοντελοποιώντας τις προτιμήσεις ως δυαδικές σχέσεις, εισάγουμε αποδοτικούς αλγορίθμους αποτίμησης των k βέλτιστων αντικειμένων.

Η έως τώρα σχετική έρευνα, αντιμετώπιζε τις εκφράσεις επί προτιμήσεων ως «μαύρα κουτιά» και εφάρμοζε την ιδέα των εξαντλητικών διαδοχικών ελέγχων υπεροχής μεταξύ των αντικειμένων μιας βάσης δεδομένων για τον προσδιορισμό των καλύτερων εξ' αυτών, γεγονός που οδηγούσε σε τετραγωνικά ως προς τον αριθμό των αντικειμένων κόστη. Αντιθέτως, εμείς υποστηρίζουμε μια προσέγγιση που

βασίζεται στη διάταξη επερωτήσεων. Η κύρια ιδέα μας βασίζεται στην εκμετάλλευση της σημασιολογίας μιας έκφρασης από προτιμήσεις, σε ό,τι αφορά τόσο τους εμπλεκόμενους τελεστές όσο και τις εμπλεκόμενες προτιμήσεις, με σκοπό τον ορισμό μιας διάταξης μεταξύ εκείνων των επερωτήσεων, των οποίων η αποτίμηση είναι αναγκαία, ώστε να ανακτηθούν τα k βέλτιστα αντικείμενα. Παρουσιάζουμε δύο πρωτότυπους αλγόριθμους, τους LBA και TBA.

Ο LBA ορίζει μια διάταξη επερωτήσεων, οι οποίες ουσιαστικά αποτελούν συζεύξεις ατομικών συνθηκών επιλογής, συμπεριλαμβάνοντας όλα τα γνωρίσματα που εμπλέκονται στις προτιμήσεις του χρήστη. Ο αλγόριθμος εξασφαλίζει ότι ο τρόπος και η σειρά ανάκτησης των αντικειμένων σέβεται τις προτιμήσεις του χρήστη, αποφεύγοντας τους ελέγχους υπεροχής, και προσπελάζοντας μόνο τα k βέλτιστα αντικείμενα, μία φορά το καθένα. Από διαφορετική οπτική, ο TBA ορίζει μια διάταξη επερωτήσεων που αποτελούν διαζεύξεις ατομικών συνθηκών επιλογής πάνω σε μοναδικά γνωρίσματα, και χρησιμοποιεί κατάλληλα κατώφλια τιμών για να σημάνει τη διακοπή της ανάκτησης των αντικειμένων, εξασφαλίζοντας ότι όλα τα εναπομείναντα αντικείμενα είναι χειρότερα των ανακτηθέντων. Εν προκειμένω, πραγματοποιούνται έλεγχοι υπεροχής μόνο για τα ήδη ανακτηθέντα αντικείμενα.

Τόσο η αναλυτική μελέτη όσο και η πειραματική αποτίμηση καταδεικνύουν την υπεροχή των αλγορίθμων που παρουσιάζουμε έναντι των υφισταμένων σε όλες τις περιπτώσεις του προβλήματος.

Επόπτης: Βασίλης Χριστοφίδης
Αναπληρωτής Καθηγητής

Στους γονείς μου Κώστα και Μαρία και στον αδερφό μου Μάνο

Ευχαριστίες

Καταρχήν θα ήθελα να ευχαριστήσω τον επόπτη μου κ. Βασίλη Χριστοφίδη για τα όσα μου προσέφερε στα δυόμισι και πλέον χρόνια της συνεργασίας μας. Χωρίς την ουσιαστική του καθοδήγηση, τη διαρκή στήριξη και τη βοήθεια που πάντα ήταν διατεθειμένος να προσφέρει, η παρούσα εργασία δεν θα μπορούσε να ολοκληρωθεί.

Επίσης ευχαριστώ τον καθηγητή κ. Ιωάννη Τζίτζικα με τον οποίο είχα την ευκαιρία να συνεργαστώ κατά την εκπόνηση της εργασίας αυτής και να αποκομίσω σημαντικές γνώσεις.

Παράλληλα θα ήθελα να ευχαριστήσω θερμά τον κ. Γεώργιο Γεωργακόπουλο για τις πολύτιμες παρατηρήσεις του οι οποίες βελτίωσαν την παρούσα εργασία.

Ένα μεγάλο ευχαριστώ αξίζει στον Περικλή Γεωργιάδη, με τον οποίο μοιραστήκαμε πολλές ώρες συζητήσεων. Η συμβολή του στην κατανόηση πολύπλοκων προβλημάτων ήταν καθοριστική ενώ η προθυμία του για βοήθεια είναι άξια αναφοράς.

Επίσης, θα ήθελα να ευχαριστήσω το Πανεπιστήμιο Κρήτης και την ομάδα Πληροφοριακών Συστημάτων του Ινστιτούτου Πληροφορικής για όσα μου προσέφεραν αυτά τα χρόνια και για τις γνώσεις που απέκτησα κατά τις σπουδές μου.

Ένα μεγάλο ευχαριστώ επίσης ανήκει σε όλους τους φίλους/ες μου με τους οποίους συνεργάστηκα καθ'όλη την διάρκεια των σπουδών μου. Τους εύχομαι ότι καλύτερο στη ζωή τους.

Τελευταίο αλλά μεγαλύτερο ευχαριστώ ανήκει στους γονείς μου Κώστα και Μαρία και στον αδερφό μου Μάνο για την αμέριστη συμπαράστασή τους σε όλες τις δυσκολίες. Για το λόγο αυτή η εργασία αυτή είναι αφιερωμένη σε αυτούς και ελπίζω να αποτελέσει μια μικρή ανταμοιβή για τις θυσίες και τις προσπάθειές τους όλον αυτό τον καιρό.

Καπανταϊδάκης Γιάννης

Contents

List of Figures

List of Tables

Chapter 1: Introduction

Preference modelling and management has attracted considerable attention in the areas of Databases, Knowledge Bases and Information Retrieval Systems in recent years. This interest stems from the fact that a rapidly growing class of untrained lay users confront vast data collections, usually through the Internet, typically lacking a clear view of either content or structure, moreover, not even having a particular object in mind. Rather, they are attempting to discover potentially useful objects, in other words, objects that best suit their preferences. A modern information system, consequently, should enable users to quickly focus on the k best object according to their preferences.

In recent years, a lot of research effort has been made for the representation of user preferences. Mainly there are two different approaches of such type of personalization, the *qualitative* ([7], [8], [9], [13], [15], [24], [30]) and the *quantitative* ([1], [14], [22], [23]). In the qualitative approach, the preferences between objects are specified directly, typically using binary relations. In the quantitative approach, preferences are specified indirectly using scoring functions that associate a numeric score with every object. An object o is preferred to an object o' if the score of o is higher than the score of o'. The qualitative approach is more powerful (in terms of expressive power) than the quantitative one, because we can model quantitatively specified preferences using preference relations, while not every (intuitively plausible) preference relation can be captured by scoring functions [8]. Moreover, there is no obvious method that the users could follow for specifying and combining scores.

In this work we confine ourselves to the qualitative approach for the representation of user preferences. More precisely, we advocate a qualitative preference framework in which users can define atomic and complex preferences as well. An atomic preference is defined as a reflexive and transitive binary relation (i.e., non-antisymmetric preorder) over the domain of an attribute. On the other hand, a complex preference is an expession that imposes

priorities over the atomic preferences by using available preference constructors (e.g. Pareto, Priorization).

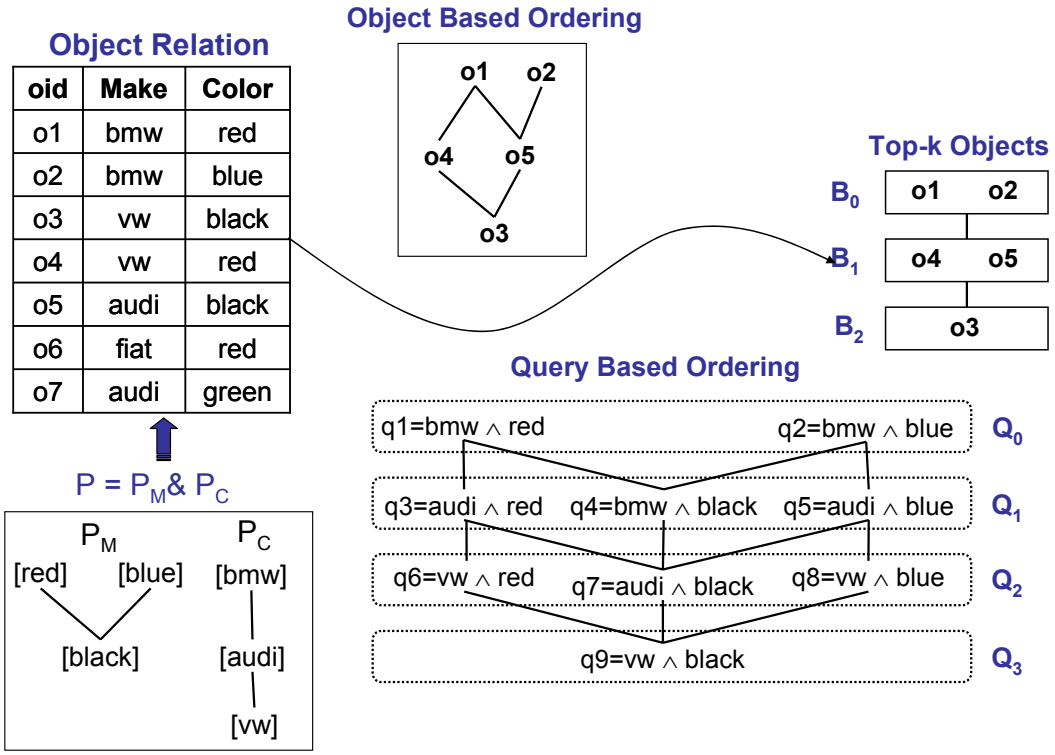

Figure 1: Evaluating the top-k objects according to qualitatively specified preferences

Assume for example an object relation $R(Make, Color)$ describing cars, as depicted in Figure 1 where for simplicity objects are identified by a oid. A user wishing to purchase a car may state that he prefers red and $blue$ cars to $black$ ones. Furthermore, he also prefers a bmw to an $audi$, and the latter to vw. Finally, he states that preferences on (M)ake (P_M) are as important as on (C)olor (P_C) ($P_M \,\&\, P_C$). Since preferences P_M and P_C are defined over individual attributes are considered as atomic preferences while $P_M \,\&\, P_C$ is a preference expression. Let us first consider the atomic preference P_M stated on the domain of the attribute Make. The domain values appearing in P_M (i.e., $bmw, audi, vw$) imply that only objects featuring the corresponding terms are of interest to the user. Furthermore, since in our example the user is not wishing to further restrict his car selection (i.e., no additional selections were made), preference P_M will partition objects of R into objects that match the

disjunction of the involved *terms* $(M = bmw) \vee (M = audi) \vee (M = vw)$ and into objects that do not (e.g., o_6). However, a preference like P_M not only partitions the matching objects according to the preference terms, but also orders the resulting partition (e.g., in decreasing order of preference) under the form of a *block sequence* (i.e., a linear order of blocks or sets). According to the database of Figure 1 and preference P_M we will have the following block sequence:

$$\{o_1, o_2\} \leftarrow \{o_5, o_7\} \leftarrow \{o_3, o_4\}$$

given that $bmw(o_1, o_2)$ precedes $audi(o_5, o_7)$ and $vw(o_3, o_4)$, and thus should be placed on the top block of objects returned to the user (note that object o_6 is filtered out). When a user preference spans more than one attributes, such as $P_M \,\&\, P_C$, we need to filter out objects by considering a disjunction of *term conjunctions* rather than atomic terms. The result of the preference $P_M \,\&\, P_C$ will thus consist of blocks of objects matching the *disjunction* of the Cartesian Product of the terms involved in P_M and P_C (o_6 and o_7 are filtered out):

$(M = bmw \wedge C = red) \vee (M = bmw \wedge C = blue) \vee$

$(M = audi \wedge C = red) \vee (M = audi \wedge C = blue) \vee (M = bmw \wedge C = black) \vee$

$(M = vw \wedge C = red) \vee (M = vw \wedge C = blue) \vee (M = audi \wedge C = black) \vee$

$(M = vw \wedge C = black)$

Then, to order this partition, we need to examine the relationship of the user preferences stated per attribute: in our example preferences P_M and P_C are considered to be of equal importance ($P_M \,\&\, P_C$). Given that *red* (o_1) or *blue bmw's* (o_2) are the most preferred ones (top block), while *black vw* (o_3) are the least preferred ones (bottom block), we obtain the following sequence of blocks (note that blocks that "tie" in terms of preferences are merged):

$$\{o_1\} \cup \{o_2\} \leftarrow \{o_4\} \cup \{o_5\} \leftarrow \{o_3\}$$

We can easily observe that not all conjunctions of preference terms will yield non empty results. It is worth noticing that the resulting linear order of blocks essentially "linearizes" the order of objects induced by the preference $P_M \,\&\, P_C$

as depicted in Figure 1. However, users usually do not wish to obtain the entire linear order of blocks but only the *top-*k objects that best suit their preferences.

In this thesis, we devise efficient top-k evaluation algorithms. Specifically, for the given user preferences our objective is to compute and deliver a linear order of n blocks (i.e., sets) of objects, where n is the smallest integer that satisfies the inequality $\sum_{i=0}^{n-1} |B_i| \geq k$. In such a linear order, each block would correspond to a screen of objects that is shown to the user, satisfying the following properties with respect to the user preferences:

a) Each block consists of non comparable objects.
b) The first block contains the most preferred objects.
c) For each block B_i other than the first and for each object in B_i there is a more preferred object in the previous block (alternatively but not equivalently, for each block B_i other than the last and for each object in B_i there is a less preferred object in the next block). The objects in $\bigcup_{i=0}^{n-1} B_i$ are called the top-k objects.

Existing algorithms ([8], [29], [30]) for the evaluation of the top-k objects according to qualitatively specified preferences, follow an *object-based ordering* approach (Figure 1). The key idea of this approach is to sequentially apply *dominance - tests* (i.e., compare two objects to determine whether one is better than the other with respect to user preferences) for every possible pair of objects. The results of these tests actually specify a preorder (i.e., a reflexive and transitive binary relation) over the objects of a relation. Subsequently, the algorithms of this approach "linearize" the preorder i.e., they turn the preorder to a linear order of blocks in a reasonable manner that respects the preorder and finally pick and deliver to the user the top-k objects. The main characteristic of the object-based ordering approach is that the flow of control of the algorithms of this family is independent of the user preferences.

Despite the wide applicability of the object-based ordering approach (since it can be used for any number of atomic preferences without indexing or sorting of

the database objects), the algorithms of this family are not appropriate for large database systems and real scale Web applications since they have serious drawbacks. An algorithm that follows the object-based ordering approach will access all objects of a relation R at least once and will perform at least one dominance test for every object in R. The total number of dominance tests that such an algorithm performs is $O(n^2)$ where n is the number of objects in R. This makes them inappropriate for large databases. Moreover, existing algorithms are inadequate for on-line (i.e., incremental) processing since the entire preorder over the objects of the relation R needs to be specified in order to return the top-k objects progressively (i.e., top-1, top-2, ..., top-k).

We advocate a *query-based ordering* approach for the evaluation of the top-k objects and we introduce two novel algorithms called *LBA* (*Lattice Based Algorithm*) and *TBA* (*Threshold Based Algorithm*) that follow this approach. Contrary to the object-based ordering approach, the flow of control of our algorithms, takes into account the preference expression given as input, as well as, the value ordering of the involved atomic preferences. The main idea of the query-based ordering approach is to use the specified user preferences for defining an ordering over queries that need to be evaluated in order to retrieve the top-k objects (see Figure 1).

In particular, LBA defines an ordering over queries which are essentially a union of conjunctions of atomic selection conditions, containing all attributes that the user preference involves. A query Q_i precedes Q_i' if the objects in the answer of Q_i (denoted by $ans(Q_i)$) are more preferred than the objects in $ans(Q_i')$. The evaluation of such a query Q_i returns the next block of the answer B_i i.e., $B_i = ans(Q_i)$. In Figure 1, according to the specified user preferences the first query that LBA will construct is the following:

$$Q_0 := \bigcup \{q_1, q_2\} \quad \text{where } q_1 := M = bmw \wedge C = red \text{ and } \quad q_2 := M = bmw \wedge C = blue$$

since the objects in $ans(Q_0)$ are clearly the top objects of R (there cannot be another object better than the objects in $ans(Q_0)$). The evaluation of Q_0 will

return the first block of the answer B_0. Nevertheless, such a query based algorithm should be also able to dynamically reformulate the queries Q_i, capturing each block when some of the partial queries q_j of Q_i return empty anwsers. To make this clear, recall again our example of Figure 1 and assume that $o_2(bmw, blue)$ was replaced by an object $o_2'(audi, blue)$. Now, the evaluation of Q_0 will return only o_1 (i.e., $ans(q_2) = \varnothing$). However, for o_2' there will not be a more preferred object in the previous block (i.e., in B_0) since o_1 is not better that o_2'. Thus, the (c) property does not hold. Therefore LBA will replace query q_2 (which results in no objects) with query $q_2' := color = 'blue' \wedge make = 'audi'$ for which it holds that $ans(q_2')$ contains the best objects of relation R that are not worse than objects in $ans(q_1)$. Q_0 will be reformed as follows: $Q_0 := \bigcup \{q_1, q_2'\}$. The evaluation of Q_0 will return o_1 and o_2'. Now for each of the remaining objects in R there is a more preferred object in the previous block (i.e., in $B_0 = \{o_1, o_2'\}$).

Notice that LBA will never perform a dominance testing over objects. The algorithm exploits user preferences and retrieves objects such that it is ensured that the objects are fetched in a way that respects user preferences. Moreover, LBA will only access the top-k objects and only once (assuming that available indexes exist). LBA is also suitable for on-line processing since it returns the next block of the answer B_i without having to compute previously the following blocks of B_i in the linear order.

However, consider a scenario where the total number of objects of a relation R is relatively very small compared to the number of distinct values that each domain contains (i.e., the selectivity of each domain value is small) or/and the number of attributes that the user preference involves is quite large. Since LBA constructs queries that are actually a combination of atomic selection conditions that contain all attributes that user preferences involve, in such a scenario LBA

will have many fruitless fetching attempts (i.e., resulting in no object) because R does not necessarily contain objects for every query that LBA will construct. Therefore in a scenario like the one described above LBA's performance is expected to drop.

For this reason we dense a second algorithm called *TBA* (*Threshold Based Algorithm*). Like LBA, TBA defines an order of queries however these queries are disjunctions of atomic selection conditions over just one attribute. As a result, TBA is expected to have less fruitless fetching attempts. Moreover TBA uses appropriate threshold values in order to determine when the fetching of objects should stop. These values work as a guarantee ensuring that objects that were not fetched are worst than the ones that were already fetched (i.e., work as an upper bound of the unseen objects). For defining the ordering of queries, TBA takes into account the selectivities of the atomic selection conditions so that to avoid fetching more objects than those actually required. However, TBA needs to perform dominance tests but only for the already retreived objects. Thus, unlike object-based ordering algorithms, TBA avoids exhaustive dominance testing among all objects which leads to quadratic costs.

In a nutshell, the contributions of this thesis are:

- We advocate a simple, yet expressive, framework for specifying qualitatively specified preferences as preorders.
- We introduce a query based ordering approach for the evaluation of the top-k objects. Unlike object-based ordering approaches, the key idea of this approach is to exploit the particular user preference semantics to define an ordering over those queries, whose evaluation is necessary for the retrieval of the top-k objects.
- Inspired by the query ordering based approach, we designed and implemented two progressive algorithms (LBA, TBA) for qualitatively specified preferences and we study their performances.
- We report the results of an extensive experimental evaluation on large datasets that shows that the algorithms that we propose outperform the existing ones under all problem instances that we tested.

The rest of this thesis is organized as follows. In Chapter 2, we introduce some preleminary material in Order Theory and present our proposed qualitative preference model. Chapter 3, fully describes the most common algorithms that have emerged so far and our novel top-k algorithms. In Chapter 4, we analyze experimentally the performance of the various top-k algorithms presented earlier. Chapter 5 discusses related work and finally, Chapter 6 summarizes our contributions and identifies issues for further research.

Chapter 2: Orders and Preferences

2.1 Introduction to Order Theory

2.1.1 Binary Relations

A binary relation R is an arbitrary association of elements of one set with elements of another (or perhaps the same) set. More specifically, a binary relation R from X to Y is a subset of the Cartesian Product $X \times Y$ (i.e., $R \subseteq X \times Y$). The statement $(x, y) \in R$ is read "*x is R-related to y*", and is denoted by xRy or $R(x, y)$. If $X = Y$ then we simply say that the binary relation is *over* X. There are several categorizations of binary relations over a set X, based on which *axioms* they satisfy. Common axioms (or relation *properties*) defined for binary relations are the following:

- reflexivity: $\forall x \in X$ it holds that xRx.
- irreflixivity: $\forall x \in X$ it holds that $\neg(xRx)$
- symmetry: $\forall x, y \in X$ it holds that if xRy then yRx
- antisymmetry: $\forall x, y \in X$ it holds that if xRy and yRx then $x = y$
- asymmetry: $\forall x, y \in X$ it holds that if xRy then $\neg(yRx)$
- transitivity : $\forall x, y, z \in X$ it holds that if xRy and yRz then xRz
- completeness : $\forall x, y \in X$ it holds that xRy or yRx (or both)

2.1.2 Orders

Certain important types of binary relations can be characterized by the axioms they satisfy. These types of relations are called *orders*. Below we present the most important orders which we intend to use in the following chapters of our work in order to formally define the model of preferences that we use[1]:

Definition 2.1: A binary relation is a *preorder*, denoted by $\preceq$, if it is reflexive and transitive. A set that is equipped with a preorder is called a *preordered set*.

Definition 2.2: A binary relation is an *equivalence relation*, denoted by $\sim$, if it is reflexive, symmetric and transitive. For an equivalence relation $\sim$ on a set X, the set of the elements of X that are related to an element, say $x \in X$, is called the *equivalence class* of element x, often denoted as $[x]$.

Definition 2.3: A binary relation which is reflexive, antisymmetric and transitive is called a *partial order* and it is denoted by $\leq$. A set with a partial order is called a *partially ordered set* or *poset*.

Definition 2.4: A binary relation is a *strict partial order*, denoted by $<$, if it is irreflexive and transitive, and therefore asymmetric.

Note that if a preorder is also antisymmetric, it becomes a partial order, whereas if it is also symmetric it becomes an equivalence relation. Let $\precsim$ be a non-antisymmetric *preorder* (i.e., a reflexive and transitive relation) over X. The *asymmetric part* of $\precsim$ is the binary relation $\prec$ over X, defined as $\forall (x, y) \in X \times X$, $x \prec y \Leftrightarrow x \precsim y \wedge \neg (y \precsim x)$. The *symmetric part* of $\precsim$ is the binary relation $\sim$ over X defined as $\forall (x, y) \in X \times X$, $x \sim y \Leftrightarrow x \precsim y \wedge y \precsim x$. It is easy to see that the asymmetric part comprises a *strict partial order* (i.e., an irreflexive, asymmetric, transitive) relation, whereas the symmetric one, an *equivalence relation* (i.e., a reflexive, symmetric, transitive relation). A *partial order* (i.e., a reflexive, antisymmetric, transitive) relation $\leq$ derives from $\precsim$

[1] The preference model that we use is an extended version of [28]

among the equivalence classes of the quotient set $X/\sim$ as follows: $[x] \leq [y] \Leftrightarrow x \precsim y$ and $[x] = [y] \Leftrightarrow x \sim y$.

For $\precsim$, being a non-antisymmetric *preorder* over X, it holds that:

- $\prec$ is transitive
- $\sim$ is transitive
- $x \sim y \wedge y \prec z \Rightarrow x \prec z$
- $x \prec y \wedge y \sim z \Rightarrow x \prec z$

For $\precsim$, being a non-antisymmetric *preorder* over X, its *asymmetric* and *symmetric parts* are disjoint and their union equals $\precsim$ (i.e., symmetry partitions $\precsim$). For any two elements x and y of a partially ordered set, if $x \leq y$ and $x \neq y$, due to antisymmetry we can write $x < y$. Similarly, for any two elements x and y of a preordered set, if $x \preceq y$ and $\neg(y \preceq x)$, we can write $x \prec y$. In either case, if $x < y$ (respectively, $x \prec y$) and there is no z such that $x < z$ and $z < y$, (respectively, there is no z such that $x \prec z$ and $z \prec y$) we will say that y is a *cover* of x, and denote it as $x \prec y$.

A partial order which is complete is called a *total (or linear) order* or a *chain*. A preorder which is complete is called a *weak order* or a *complete preorder*.

Elements x and y of a set X, for which it holds that xRy or yRx are said to be *comparable*; otherwise, x and y are *incomparable*. More formally, we define:

Definition 2.5: Given a relation R over a set X, the *incomparability relation* (usually denoted as $\|$ when R is some order), is defined as the *complement relation* R^c over the same set X; i.e., xR^cy, iff $\neg(xRy) \wedge \neg(yRx)$.

For example, $x \| y$ means that elements x and y are *incomparable* to each other (i.e., none of the relations xRy and yRx hold). Note that the above terminology may be misleading when R is a strict partial order, as its

complement R^c may capture two very different situations: either *incomparability* indeed, or *comparable equality*; only when R is a preorder or a partial order the term *incomparability* have its literal meaning.

2.1.3 Graphical Representation of Partial Orders

Any relation R over a (finite) set X may be visually represented by a directed graph (V, E), with a bijective mapping of the elements of X onto the vertices of V and a bijective mapping of the pairs of R onto the edges of E.

A graph of a partial order (or a preorder, accordingly) would be very "busy", carrying a lot of redundant information: self-loops (v, v) for *every* node, deriving from reflexivity, as well as transitive edges (v_1, v_3), with both (v_1, v_2) and (v_2, v_3) being present. Furthermore, one may make two more observations: antisymmetry ensures that in such a graph there could not be any two vertices v_1 and v_2 with both edges (v_1, v_2) and (v_2, v_1) present; in conjunction with transitivity, antisymmetry also forbids any longer loop, meaning that the graph, with the exception of self-loops, has one and only direction. Exploiting the above, a partial order may be graphically represented by a *Hasse diagram*. Before we formally define a Hasse diagram we need to introduce the following auxiliary definitions:

Definition 2.6: The *transitive closure* of a binary relation R on a set X is the minimal transitive relation R' on X that contains R.

Definition 2.7: The *reflexive closure* of a binary relation R on a set X is the minimal reflexive relation R' on X that contains R.

Definition 2.8: The *transitive reduction* of a binary relation R on a set X is the minimum relation R' on X with the same transitive closure as R.

Definition 2.9: The *reflexive reduction* of a binary relation R on a set X is the minimum relation R' on X with the same reflexive closure as R.

Now we can proceed to the formal definition of a Hasse diagram:

Definition 2.10: A *Hasse diagram* of a partial order is a directed acyclic graph of its *reflexive and transitive reduction*, where direction is omitted, as it is implied by the diagram's upward orientation.

A Hasse diagram may also depict a neither symmetric, nor antisymmetric preorder. In this case it essentially represents the *partial* order of the equivalence classes of the quotient set $X / \sim$, rather than the preorder itself. The Hasse diagram of an equivalence relation is simply a set of non-connected nodes, each of which is a representative of an equivalence class. So, in all cases above, the Hasse diagram obeys conventions of what each nodes stands for (class representatives in some cases) and the only case where it directly depicts a relation is the case of a *strict* partial order. Note that, in all cases, all lines in a Hasse diagram correspond to the *cover relation* $\prec$ (i.e., the transitive reflexive reduction of a partial order), reflecting on the strict part $<$ of the partial order $\leq$.

Example 2.1: Let us assume the Hasse diagram for a set $X = \{a, b, c, d, e, f\}$, a preorder $\preceq$ over X, with $d \preceq b$, $b \preceq a$, $d \preceq a$, $c \preceq a$, $d \preceq f$ and $f \preceq d$. Such a diagram cannot represent the preorder itself directly, so, as discussed above, it will depict the *partial* order of the equivalence classes of the quotient set $X / \sim$. There are five equivalence classes $[a] = \{a\}$, $[b] = \{b\}$, $[c] = \{c\}$, $[d] = \{d, f\}$, $[e] = \{e\}$, and let's choose a single representative from each one to use in the diagram. The resulting Hasse diagram is illustrated in Figure 2.

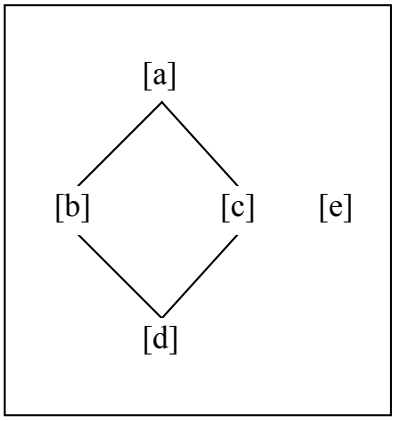

Figure 2: The Hasse diagram of $\preceq$ over $X / \sim$

2.1.4 Notions on Posets

In a partially ordered set there are some elements that play a special role. The most basic examples are given by the *maximal* and the *minimal* elements of a poset.

Definition 2.11: Let a partial order $\leq$ over a set of elements X. An element $x \in X$ is a *maximal element* of $\leq$, if $\neg\exists x' \in X$ such that $x \leq x'$.

Definition 2.12: Let a partial order $\leq$ over a set of elements X. An element $x \in X$ is a *minimal element* of $\leq$, if $\neg\exists x' \in X$ such that $x' \leq x$.

We may partition the elements of a partial order relation X into non-overlapping parts called *blocks* (or *layers* or *buckets*) that cover all of X using various topological criteria. To define our approach formally we need some auxiliary definitions that we adapt from [28]:

Definition 2.13: Let us call *path* from an element x to an element x' of a partial order $\leq$, any sequence of pairs of the form $(x, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n), (x_n, x')$ such that $x \leq x_1, x_n \leq x'$ and $x_{i-1} \leq x_i$ for $i = 2 \ldots n$. The integer $n+1$ is called the *length* of the path, and it is clear that there may be zero, one or more paths from x to x'.

Now, assume that B_0 contains all elements that are maximal (or minimal) with respect to $\leq$. The definition of each other block B_i relies on the notion of *distance* of an element from B_0.

Definition 2.14: The *distance* of an element x_0 from B_0 is defined to be the length of the longest path from an element x to the element x_0, when x ranges over all elements of B_0.

Block B_i is defined to be the set of all elements that are at distance i from B_0. Note that if x_0 belongs to B_0, then its distance is defined to be equal to 0. It is easy to see that elements of the same block are incomparable to each other (otherwise they wouldn't have the same distance from B_0).

2.1.5 From Partial Order of Elements to Linear Order of Blocks

Let $\leq$ be a binary order relation (e.g., a preorder, a partial order, or a strict partial order) on a set $O \neq \varnothing$ and let 2^O be the *powerset* of O minus $\varnothing$. In this section, we define relations over subsets of O, i.e., over 2^O, which derive from the initial order $\leq$ over O.

Definition 2.15: Assuming that $X, Y \in 2^O$ for which does not necesserily hold $X \cap Y = \varnothing$, we define the following relations over 2^O :

- Let[2] $X \leq_{\forall\exists} Y$, iff $\forall x \in X$, $\exists y \in Y$ such that $x \leq y$
- Let $X \leq_{\exists\forall} Y$, iff $\forall y \in Y$, $\exists x \in X$ such that $x \leq y$

Note, that apart from transitivity, which is trivial to prove, whether other order axioms hold in each of these relations over sets, depends on the nature of the initial poset and probably other assumptions, and need to be proved, thus the use of the term *set order*, instead of *set relation* may be abusive. Let $B_0, B_1, \ldots, B_n$[3] a sequence of blocks of O that were produced as described in previous section.

Theorem 2.1: If B_0 contains the maximal elements of $\leq$, a $\leq_{\forall\exists}$ relation is defined between blocks $B_0, B_1, \ldots, B_n$ (i.e., $B_n \leq_{\forall\exists} B_{n-1} \leq_{\forall\exists} \ldots \leq_{\forall\exists} B_0$).

Proof 2.1: For every element x_i in B_i there is a longest path p from some element of B_0 to x_i. Let x_i' be the predecessor of x_i in p (i.e., $x_i \leq x_i'$). Clearly, the sub-path of p ending in x_i' is the longest path from B_0 to x_i' (otherwise, p is not the longest path to x_i thus a contradiction). It follows that x_i' is in B_{i-1} and that $x_i \leq x_i'$.

---

[2] As a rule of thumb, the first quantifier runs on the left operand set, the second on the right, and the outer quantifier is denoted by the line above it.

[3] For each of these blocks B_i it holds $B_i \in 2^O$.

Theorem 2.2: If B_0 contains the minimal elements of $\leq$, a $\leq_{\exists\overline{\forall}}$ relation is defined between blocks $B_0, B_1, \ldots, B_n$ (i.e., $B_0 \leq_{\exists\overline{\forall}} B_1 \leq_{\exists\overline{\forall}} \ldots \leq_{\exists\overline{\forall}} B_n$).

Proof 2.2: For every element x_i in B_i there is a longest path p from some element of B_0 to x_i. Let x_i' be the predecessor of x_i in p. The sub-path of p ending in x_i' is the longest path from B_0 to x_i' (otherwise, p is not the longest path to x_i thus a contradiction). It follows that x_i' is in B_{i-1} and that $x_i' \leq x_i$.

Theorem 2.3: $\leq_{\overline{\forall}\exists}$ relation defines a linear order of blocks.

Proof 2.3: Clearly $\leq_{\overline{\forall}\exists}$ is reflexive and transitive (since $\leq$ is reflexive and transitive). Moreover, since each block consists of mutually incomparable elements it is also antisymmetric and thus is a partial order. Due to the definition "$B_i \leq_{\overline{\forall}\exists} B_j$ iff $\forall x_i \in B_i$, $\exists x_j \in B_j$ such that $x_i \leq x_j$", we have $B_n \leq_{\overline{\forall}\exists} B_{n-1} \leq_{\overline{\forall}\exists} \ldots \leq_{\overline{\forall}\exists} B_0$. As a result, $\leq_{\overline{\forall}\exists}$ actually defines a linear order between blocks.

Similarly, we can prove that $\leq_{\exists\overline{\forall}}$ also defines a linear order of blocks. Therefore since $\leq_{\overline{\forall}\exists}$ and $\leq_{\exists\overline{\forall}}$ relations define linear orders of blocks from now on we could write $<_{\overline{\forall}\exists}$ and $<_{\exists\overline{\forall}}$ to denote those linear orders.

Example 2.2: Figure 3 illustrates the two individual orderings $<_{\overline{\forall}\exists}$ and $<_{\exists\overline{\forall}}$ between the blocks B_0, B_1, B_2 of a partially ordered set O. $<_{\overline{\forall}\exists}$ ordering occurs if we use as basic block B_0 the block that contains the maximal elements of O and $<_{\exists\overline{\forall}}$ occurs if we use as B_0 the block that contains the minimal elements.

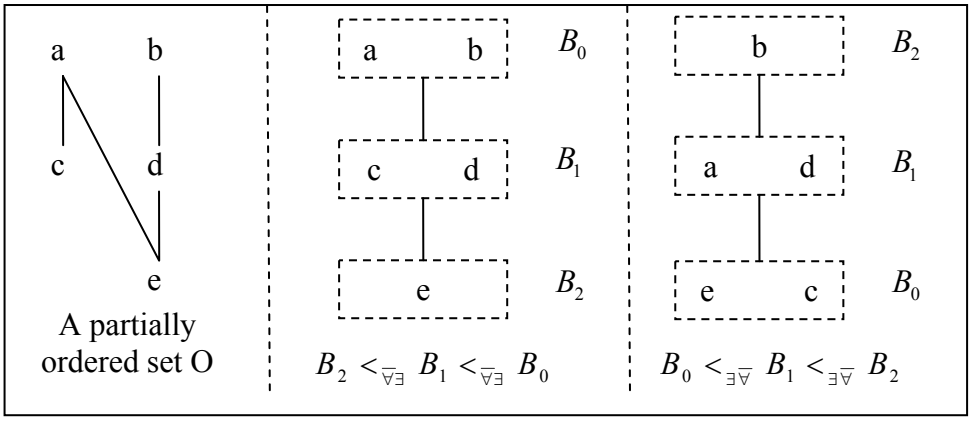

Figure 3: The $<_{\overline{\forall\exists}}$, $<_{\exists\overline{\forall}}$ ordering of blocks B_0, B_1, B_2 of a partially ordered set O

2.2 Qualitative Preference Model

Let $R(\mathcal{A})$ denote a relation scheme, where R is the name of the relation and $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ is a set of *attribute names* with associated domains $dom(A_i)$. Without loss of generality, we assume the attribute domains pair-wise disjoint, i.e., $dom(A_i) \cap dom(A_j) = \varnothing$ for every $i \neq j \in [1 \ldots n]$. As null values are possible, in order to keep notations simple, we use $dom(A_i)$ to denote $dom(A_i) \cup \{\perp\}$, where "$\perp$" stands for the null value. We shall also use the notation $dom(A) = dom(A_1) \times \ldots \times dom(A_m)$ and $dom(A_\cup) = dom(A_1) \cup \ldots \cup dom(A_m)$ to denote the Cartesian Product and the union of domains, of a non empty set of attributes $A \subseteq \mathcal{A}$. An object over a scheme $R(\mathcal{A})$ associates with each $A_i \in \mathcal{A}$ a value taken from its domain. As usual $o[A]$ denotes the projection of an object o onto a non empty set of attributes $A \subseteq \mathcal{A}$. A relation R over the scheme $R(\mathcal{A})$ (also called an *instance*) is a finite set of objects o such that $o[\mathcal{A}] \in dom(\mathcal{A})$.

2.2.1 User Preferences

In order to proceed to the general preference definition we should take into consideration two important factors:

- The user, most times, is not in position to know the objects that a database contains. Thus, the preferences should be defined on structures of information that are not influenced by the available objects. Such structures are the attribute domains.
- The number and the nature of attributes that are involved in a preference expression vary. Therefore, the definition of a preference should be based on the attributes that the preference involves.

Definition 2.16: Let us assume a relation scheme $R(\mathcal{A})$. A *preference* P_A over a non empty set of attributes $A = \{A_1 \dots A_m\} \in 2^{\mathcal{A}} \setminus \varnothing$ is a non-antisymmetric partial preorder over $dom(A) = dom(A_1) \times \dots \times dom(A_m)$[4] denoted as $P_A = (dom(A), \precsim_{P_A})$, where $\precsim_{P_A} \subseteq dom(A) \times dom(A)$. For m-tuples $v, v' \in dom(A)$, $v \precsim_{P_A} v'$ is interpreted as v *is at most as preferable as* v' (or equivalently, v' *is at least as preferable as* v).

We shall pronounce those v', v for which both $v \precsim_{P_A} v'$ and $v' \precsim_{P_A} v$ hold, as *equally preferred* or *indistinguishable w.r.t. preference* P_A. As symmetry holds by definition and reflexivity with transitivity are inherited from $\precsim_{P_A}$, the *preference equality* relation is an equivalence relation, i.e., equally preferred tuples v', v are *equivalent,* or belong to the same equivalence class, thus we will denote this relation as $v' \sim_{P_A} v$.

If $v \precsim_{P_A} v'$ but $\neg(v' \precsim_{P_A} v)$, we can write $v \prec_{P_A} v'$ which is interpreted as v' *is (strictly) more preferable than* v. As asymmetry and irreflexivity holds by definition and transitivity is inherited from $\precsim_{P_A}$, the asymmetric part $\prec_{P_A}$ of $\precsim_{P_A}$ comprises a *strict partial order* relation. If neither $v \precsim_{P_A} v'$ nor $v' \precsim_{P_A} v$ hold, then we will say that v', v are *incomparable* and we will write $v \|_{P_A} v'$. The *incomparability* relation carries symmetry, by definition, but apart from it, it satisfies no other order axioms in general.

[4] The order of factors within the Cartesian Product is considered of no particular significance.

When A is a trivial single-factor Cartesian Product (i.e., $A = \{A_i\}$) we will call $P_i = (dom(A_i), \precsim_{P_i})$ an *atomic preference* over A_i, where $dom(A_i)$ is the domain of A_i.

For a preference $P_A = (dom(A), \precsim_{P_A})$, the values of domain $dom(A)$ can be separated in two concrete categories, proportionally whether they take active part or no in the preorder $\precsim_{P_A}$.

Definition 2.17: Given a preference $P_A = (dom(A), \precsim_{P_A})$, a value $v \in dom(A)$ that is not involved in the partial preorder relation in any other way except though reflexivity (i.e., $\neg\exists v' \in dom(A)$, $v' \neq v$, such that $v \precsim_{P_A} v'$ or $v' \precsim_{P_A} v$) will be called *inactive* (otherwise, it will be called *active*) and clearly it is incomparable to all other values of $dom(A)$.

We denote $V(P_A, A)$ the set of active values of $dom(A)$ according to P_A and $V^c(P_A, A)$ the set of inactive values, respectively. Notice that:

- $V(P_A, A) = \times_{i=1}^{m} V(P_{Ai}, A_i)$ where $V(P_{Ai}, A_i)$ denotes the set of active values from the domain of A_i appearing in an atomic preference P_{Ai}
- $V(P_A, A) \cup V^c(P_A, A) = dom(A)$
- $V(P_A, A) \cap V^c(P_A, A) = \varnothing$

We make this seperation since inactive values actually don't take part in the ordering $\precsim_{P_A}$, creating, thus, a sense of "indifference" of the user to each inactive value. In essence, there is no need to take inactive values into account since only active values have interest to a particular user and need to be specified (regardless of whether they are actually instantiated in R).

Next we discuss preference expressions, which capture the semantics of combining or synthesizing user preferences over more than one individual attributes.

Definition 2.18: Let P_i be an atomic preference over any individual attribute A_i of a relation R. A *preference expression* P is defined as:

$$P = P_i \mid (P \,\&\, P_i) \mid (P_i \,\&\, P) \mid (P \rhd P_i) \mid (P_i \rhd P)$$

It should be stressed out that an atomic preference appears only once in a preference expression. Semantically, this interprets a current limitation in this work; we are able to capture the relative importance among different preferences, where each is provided and mentioned only once. On the other hand, this complies with the fact that values in the active preference domains also appear only once in the respective preferences. A preference expression P_A, as defined above, spans over the set A of attributes it involves, so we are able to use $V(P_A, A)$ as defined earlier. The binary preference operators & and $\rhd$ define the *pareto* and *proritized* composition operations on two preferences as follows.

Definition 2.19: Preference $P_A = (dom(A), \precsim_{P_A})$ is the *pareto preference* (denoted by $P_A = P_1 \,\&\, \ldots \,\&\, P_m$) of m atomic preferences $P_i = (dom(A_i), \precsim_{P_i})$ defined over A when $\forall v = (v_1 \ldots v_m), v' = (v_1' \ldots v_m') \in dom(A)$ we have:

- $v' \prec_{\&} v$: iff $\exists i \in [1 \ldots m]$, such that $v_i' \prec_{P_i} v_i$ and $\forall j \in [1 \ldots m] - \{i\}$ it holds $v_j' \precsim_{P_j} v_j$
- $v' \sim_{\&} v$: iff $\forall i \in [1 \ldots m]$ it holds $v_i' \sim_{P_i} v_i$
- $v' \|_{\&} v$ in all other cases

Definition 2.20: Preference $P_A = (dom(A), \precsim_{P_A})$ is the *prioritized preference*, (denoted by $P_A = P_1 \rhd \ldots \rhd P_m$) of m atomic preferences $P_i = (dom(A_i), \precsim_{P_i})$ defined over A when $\forall v = (v_1 \ldots v_m), v' = (v_1' \ldots v_m') \in dom(A)$ we have:

- $v' \prec_{\rhd} v$: iff $\exists i \in [1 \ldots m]$ such that $v_i' \prec_{P_i} v_i$ and $\forall j \in [1 \ldots i-1]$ it holds $v_j \sim_{P_j} v_j'$

- $v' \sim_{\triangleright} v$: iff $\forall i \in [1 \ldots m]$ it holds $v'_i \sim_{P_i} v_i$
- $v' \|_{\triangleright} v$ in all other cases

Note that a pareto preference is a combination of *mutually non dominating* preferences. On the other hand, a prioritized preference treats P_1 as *more important* than preference P_2, which in turn is more important than P_3, etc., up to preference P_m.

Both operators are associative, allowing us to apply each of then on more than two operands, without using parentheses; the Pareto operator is commutative, but not the Prioritization one, whereas neither of the two is distributive over the other. Let us, also, assume that when omitting parentheses, the operators take priority from left to right.

Example 2.3: Consider, for instance, the relation schema $R(A,B,C)$ where the domain of attributes is given respectively by the sets $dom(A) = \{a_1, a_2, a_3\}$, $dom(B) = \{b_1, b_2, b_3\}$, $dom(C) = \{c_1, c_2, c_3\}$. Also suppose that a user has defined the atomic preferences $P_1 = (dom(A), \precsim_{P_1})$, $P_2 = (dom(B), \precsim_{P_2})$, $P_3 = (dom(C), \precsim_{P_3})$ such that $a_3 \precsim_{P_1} a_1$, $a_3 \precsim_{P_1} a_2$, $b_2 \precsim_{P_2} b_1$, $b_3 \precsim_{P_2} b_2$, $c_2 \precsim_{P_3} c_1$, $c_3 \precsim_{P_3} c_2$. Notice that all values here are active. Figure 4 depicts their corresponding Hasse diagrams[5]:

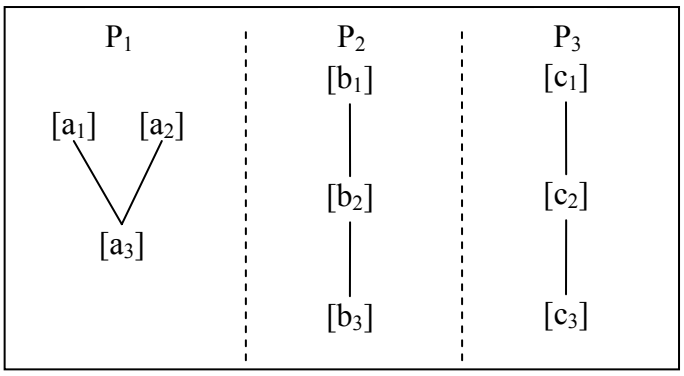

Figure 4: Hasse diagram for three atomic preferences

[5] Recall that the above *Hasse diagram* represents the *partial* order of the equivalence classes of the quotient sets.

Now, consider two elements (a_3, b_1, c_1), $(a_1, b_1, c_3) \in dom(A) \times dom(B) \times dom(C)$. According to the above definitions we have:

1. $(a_3, b_1) \prec_{P_1 \& P_2} (a_1, b_1)$ due to $a_3 \prec_{P_1} a_1$ (i.e., since $a_3 \preceq_{P_1} a_1$ and $a_3 \not\sim a_1$) and $b_1 \preceq_{P_2} b_1$

2. $(a_3, b_1, c_1) \|_{P_1 \& P_2 \& P_3} (a_1, b_1, c_3)$ because $a_3 \prec_{P_1} a_1$ but $c_3 \prec_{P_3} c_1$

3. $(a_1, b_1, c_3) \prec_{P_2 \rhd P_3 \rhd P_1} (a_3, b_1, c_1)$ as both contain b_1 and because P_3 is prioritized to P_1

4. $(a_3, b_1, c_1) \prec_{(P_1 \& P_2) \rhd P_3} (a_1, b_1, c_3)$ due to (1) and since P_3 is the least important

2.2.2 From Tuple to Object Ordering

Given a preference $P_A = (dom(A), \preceq_{P_A})$, we can use the definitions above to *infer* a non-antisymmetric partial preorder of the objects themselves in a relation R through projection, as follows:

$$\forall o, o' \in R,\ o \preceq_{P_A} o' \textit{ iff } o[A] \preceq_{P_A} o'[A]$$

$o \preceq_{P_A} o'$ means that o *is at most as preferable as* o', (or equivalenty, o' *is at least as preferable as* o). The process of comparing two objects in order to decide which one is more preferred than the other is referred in the literature as "*dominance testing*". If both $o \preceq_{P_A} o'$ and $o' \preceq_{P_A} o$ hold, we shall pronounce those objects o', o as *equally preferred* and we will denote this relation as $o' \sim_{P_A} o$ (i.e., belong to the same equivalence class). If $o \preceq_{P_A} o'$ but $\neg(o' \preceq_{P_A} o)$, we can write $o \prec_{P_A} o'$ which is interpreted as o' *is (strictly) more preferable than* o or o' *dominates* o. Finally if neither $o \preceq_{P_A} o'$ nor $o' \preceq_{P_A} o$ hold, then we will say that o', o are *incomparable* and we will write $o \|_{P_A} o'$.

Now let $\leq$ be the partial order relation which derives from $\precsim_{P_A}$ among the equivalence classes of the quotient set $R/\sim_{P_A}$. $[o] \leq [o']$ will mean that the equivalence class of o is at most as preferable as the equivalence class of o'. However if $[o] \leq [o']$ and $[o] \neq [o']$ we can write $[o] < [o']$ meaning that the user prefers object o' (or any object equivalent to o') to object o (or any object equivalent to o).

Example 2.4: Assume preference $P = P_1 \& P_2$ where $P_1 = (dom(A), \precsim_{P_1})$, $P_2 = (dom(B), \precsim_{P_2})$ are the atomic preferences of Figure 4 and the following relation R:

oid	A	B	C
o_1	a_1	b_1	c_1
o_2	a_3	b_1	c_3
o_3	a_3	b_1	c_2
o_4	a_1	b_2	c_1

Figure 5: A relation R

According to the definitions described above we will have the following relations over the objects of R: $o_2 \prec_P o_1$, $o_3 \prec_P o_1$, $o_4 \prec_P o_1$, $o_2 \sim_P o_3$, $o_2 \|_P o_4$, $o_3 \|_P o_4$. As a result, there are three equivalence classes $[o_1] = \{o_1\}$, $[o_2] = \{o_2, o_3\}$, $[o_4] = \{o_4\}$ defined. The resulting Hasse diagram is illustrated in Figure 6.

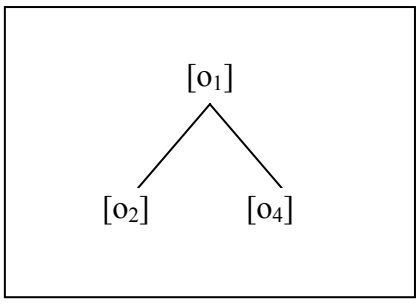

Figure 6: The Hasse diagram of $\precsim_P$ over R

Now, given a preference P over A in a relation R, we shall call *active* those objects $o \in R$ which contain active values over every attribute in A[6] while the rest of the objects are called *inactive*; Active objects, denoted as $Act(P, A)$ are those that represent items which are interesting to the user, as they contain the combinations of interesting attribute values. Moreover, active objects can be ordered with respect to others, while inactive ones are those that cannot be ordered.

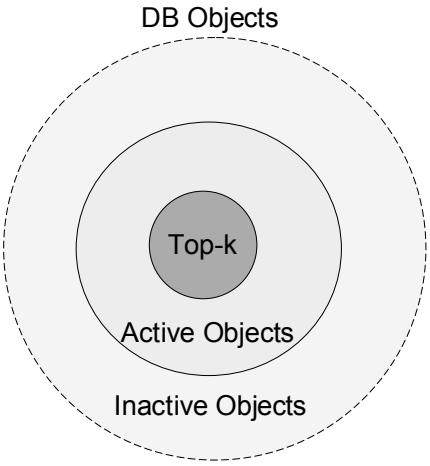

Figure 7: Active and inactive objects

For example assume that the relation R of Figure 5 contains one more object $o_5(a_4, b_1, c_1)$. Since $a_4 \notin V(P_1, A)$, o_5 is considered as inactive object and clearly it cannot be ordered with respect to the remaining objects of R. In existing frameworks ([7], [8], [9], [13], [15], [24], [30]), inactive objects are considered as incomparable to the active ones and thus returned in the first block of the result (as undominated). For instance, o_5 will be returned in the same block as object o_1, although the user definitely prefers the latter. As a consequence, our approach partitions objects of R into active and inactive ones and relies only on the active objects to retrieve the top-k objects of a relation R (Figure 7).

[6] $o[A] \in V(P_A, A)$

Chapter 3: Top-*k* Algorithms

In the previous chapter we have shown how from a given preference $P = (dom(A), \precsim_P)$, we can infer a partial order $\leq$ (which derives from $\precsim_P$) among the equivalence classes of the quotient set $R/\sim_P$. Furthermore, we have seen how to form a linear order of mutually disjoint blocks of classes of objects that respects $\leq$ and as a result the initial user preferences. In this linear order each block would correspond to a screen of incomparable equivalence classes that is shown to the user, with the most preferred classes of objects appearing first. Nevertheless, the presence of equivalent and of incomparable objects leaves space for more than one different orderings of R. In particular we have considered two linear orders of blocks ($<_{\exists\forall}, <_{\forall\exists}$) that actually satisfy the above requirements.

Let us assume a $<_{\forall\exists}$ ordering between the blocks of the answer. In order to retrieve the most preferred objects of the partial order, all the succeding blocks have to be previously computed since a $<_{\forall\exists}$ order imposes a "down to top" orientation. Thus, in the general case that we assume where the number of available active objects in relation R is large and the number k is small, a $<_{\forall\exists}$ ordering is not suitable due to the expensive computation cost and its lack of progressiveness (we actually have to order the entire relation R). As a result of this observation we mainly focus on the $<_{\exists\forall}$ ordering.

For any given user preference $P = (dom(A), \precsim_P)$ and an integer k, our purpose is to provide efficient evaluation algorithms for computing the top-k objects of R.

Problem Statement: Given a relation R our objective is for every possible $P = (dom(A), \precsim_P)$ and k parameter to compute and deliver to the user a linear order of n blocks of equivalence classes of objects $B_0, B_1, \ldots, B_{n-1}$, where n is

the smallest integer that satisfies the inequality $\sum_{i=0}^{n-1}|B_i| \geq k$ and $|B_i|$ is the cardinality of block B_i in objects. In such a linear order, each block would correspond to a screen of equivalence classes of objects that is shown to the user, satisfying the following properties:

1. $\forall [o],[o'] \in B_i$ it holds $[o] \| [o']$[7]
2. $\forall [o] \in B_0, \neg\exists [o'] \notin B_0$ such that $[o] \leq [o']$
3. $\forall i \in [0 \ldots n-1], j \in [i \ldots n]$ it holds $B_j <_{\forall\exists} B_i$ (i.e., $\forall [o] \in B_j$, $\exists [o'] \in B_i$ such that $[o] \leq [o']$)

where $\leq$ is the induced partial order from $\precsim_P$ among the equivalence classes of the quotient set $R/\sim_P$ and $\|$ its incomparability relation. The elements in $\bigcup_{i=0}^{n-1} B_i$ are called the top-k objects.

Note that according to the definitions above $\sum_{i=0}^{n-1}|B_i|$ can be greater than k. In that case the user is able to select between retrieving all objects in $B_0, B_1, \ldots, B_{n-1}$ or to stop the presentation of objects after showing the k^{th} object (this can happen before all objects of a block have been shown to the user). In either case we shall denote as $q_{k,P}(R)$ the set that contains the top-k objects of R according to P.

In this chapter we present two broad approaches that can be applied to tackle the problem at hand, namely the *object-based* and the *query-based* approach. Section 3.1 details the object-based ordering approach that has emerged so far and the most common algorithms (i.e., BNL, Best) that follow this approach. In section 3.2 we present a novel (to the best of our knowledge) query-based ordering approach and present two query-based top-k algorithms.

[7] Abusing notation we are able to generalize a relation R on objects to a relation on classes (or sets) of objects as follows: $[o]\, R\, [o']$ iff $\forall o \in [o]\, \forall o' \in [o']\, o\, R\, o'$.

3.1 Object-based ordering

For the computation of the top-k objects of a relation R, a relational operator that has been variously called *winnow* ([7], [8]), *Best* [30] or *BMO* ([13], [15]) has been introduced. Winnow selects the set of the most preferred objects (i.e., the first block), according to a given preference expression $P = (dom(A), \precsim_P)$. For the evaluation of the winnow operator two basic algorithms *Block Nested Loop* (*BNL*) [29] and *Best* [30] have been introduced. The core element of these algorithms is dominance - testing. They essentially iterativelly eliminate every object o, for which there is a dominating object o' such that $o \prec_P o'$. These algorithms can be also extended to produce the top-k results matching a preference expression, as follows: *If the result res of the winnow operator has m objects and* $m \geq k$*, return them. Otherwise deliver these m objects (i.e., return the first block) and for finding the remaining ones winnow is called again over* $R \setminus res$. So, to obtain the top-k objects a number of iterations need to be performed (in the worst case we will have k iterations). The main characteristic of the object-based ordering algorithms is that they are agnostic of the preference expressions. As a matter of fact, user preferences are treated as a black box by the dominance test. In the following sections we fully describe algorithms BNL and Best that are used for the evaluation of winnow operator.

3.1.1 Block Nested Loop (BNL)

BNL algorithm [29] (Figure 8) repeatedly reads the object relation R. The idea of this algorithm is to keep a *window* W in main memory of the best equivalence classes of objects discovered so far. All the classes of objects in the window are incomparable and they all need to be memorized, since each may turn out to dominate some input objects processed in a later step. When an object o is read from R and it is active[8] (line 5), it is compared to a representative o' from all classes of the window (line 7) and, based on this comparison, o is either eliminated or placed into the window or when there is

[8] An object is active according to a preference $P = (dom(A), \precsim_P)$ iff $o[A] \in V(P, A)$

no space into a temporary table *Temp* which will be considered in the next iteration step of the algorithm. For any active object o four cases can occur:

- o is less preferable than a representative from each equivalence class within the window (line 8). In this case, o is eliminated and will not be considered in the current iteration. Of course, o need not be compared to all class representatives of the window in this case.
- o is more preferable than one or more representatives in the window (line 9). In this case, these equivalence classes are eliminated; that is, these classes are removed from the window.
- o is equivalent with a window representative. If there is enough room in the window, o is inserted into the corresponding class (line 12) $[o']$. Otherwise, $[o']$ is removed from the window, is inserted to a temporary table *Temp* on disk and then o is inserted into $[o']$ (lines 13-15). o need not be compared to the remaining class representatives of the window in this case.
- o is incomparable with all representatives in the window. In that case o defines a new equivalence class $[o]$. If there is enough room in the window, $[o]$ is inserted into the window (lines 17, 18). Otherwise, o is inserted to a temporary table *Temp* on disk (line 20). The objects of the temporary table will be further processed in the next iteration step of the algorithm (line 21).

Initially, the first object will naturally be put into the window because the window is empty. At the end of each iteration, BNL can only output the classes of the window for which their representative has been compared to all objects that have been written to the temporary table; these classes contain objects that are not dominated by any other (i.e., they are top-k objects). Specifically, BNL outputs and ignores for further processing those classes which were inserted into the window when the temporary table was empty (line 21). These classes are guaranteed to be in the next block of the answer B_i since they have been compared to all other objects that were put into *Temp*. Therefore BNL marks

(line 19) all classes that were inserted into the window when *Temp* was empty. The remaining classes of W must be compared against those stored in the temporary table. Thus, BNL has to be executed again, this time using *Temp* as input, until there are no remaining classes in *Temp*. When the temporary table is empty (line 4), BNL has found all the objects that belong in the next block B_i. If the number of the returned objects is more than k the algorithms stops (line 24). Otherwise BNL is executed again over $R \setminus B_i$ (line 23) in order to find the next block B_{i+1} until the number of the returned objects exceeds k.

Block Nested Loop

input: a relation R, a preference expression P, an integer k

output: the $<_{\overline{\forall\exists}}$-aware top-$k$ objects of R according to P

1: $i = 0, \text{Result} = Temp = Input = W = B_i = \varnothing$ //W keeps the best objects discovered

2: Repeat

3: $Input = R$

4: While $Input \neq \varnothing$:

5: For each active object $o \in Input$ do:

6: Dominated = false

7: While (not (Dominated)) $\wedge\, \exists o' \in W$ not compared with o:

8: If $o \prec_P o'$ then Dominated = true

9: ElseIf $o' \prec_P o$ then remove $[o']$ from W

10: ElseIf $o' \sim_P o$ then

11: If MemoryAvailable then

12: $[o'] = [o'] \cup o$; stop comparisons for o

13: Else remove $[o']$ from W

14: $Temp = Temp \cup [o']$

15: $[o'] = [o'] \cup o$; stop comparisons for o

16: If not(Dominated) then

17: If MemoryAvailable then

18: $W = W \cup [o]$

19: If $Temp = \varnothing$ then $mark([o])$

20: Else $Temp = Temp \cup [o]$

21: $Input = Temp,\ B_i = B_i \cup \{[o] \in W \mid mark([o])\}$

22: $return\ B_i,\ |result| = |result| + |B_i|$

23: If |result| $< k$ then $R = R \setminus B_i,\ i = i + 1$

24: $Until$ |result| $\geq k$

Figure 8: Block Nested Loop Algorithm

The main advantage of BNL is its simplicity, since it can be used without indexing or sorting the input relation R. However, it is sensible to the amount of the available main memory. A small memory may lead to numerous iterations while in case where the size of an equivalence class in W exceeds the size of W then the algorithm can not terminate. BNL requires to access all objects of a relation R at least once and to perform at least one dominance test for every active object in R. This makes BNL inappropriate for large databases. Another disadvantage of BNL is its inadequacy for on-line processing since it has to read the entire data relation before it returns the first block B_0 (line 3).

Best case time complexity: In the best case the result (i.e., all blocks of the answer) fits into the window and the algorithm terminates in one iteration. Therefore the best case time complexity of BNL is $O(n)$ where n is the number of objects in R.

Worst case time complexity: The worst case time complexity of BNL is $O(n^2)$ and occurs when a block of the answer is very large compared to the amount of the available memory (e.g., all objects in R are incomparable to each other).

Space Complexity: The memory requirements of BNL depend on the size of window W and not on the size of relation R. Therefore we can write that the space complexity of BNL is $O(1)$.

3.1.2 Best

Like BNL, Best [30] (Figure 9) is executed in several iteration steps. Each step consists of a one or more scans over a set of candidate objects which might belong to the output B_i of the i^{th} step. The main difference between BNL and Best is that the latter tries to restrict the search space of the input relation R as much as possible for the subsequent iterations of the algorithm[9]. In order to achieve that, Best keeps in memory for each object o a set $D_{\succ}^{o}$ that contains all objects which have been compared to o and have been dominated by it. However, Best does not suffer by bounded memory requirements as BNL does.

When an active object o' read from the input R is compared with an object o_S which is kept in main memory and temporarily plays the role of the *selected* object. Object o_S is a representative of an equivalence class S_i containing some of the best objects discovered so far. For any active object o' four cases are possible:

- $o' \sim_P o_S$ in this case o' is added to S_i and o_S remains the selected object
  (line 8).
- $o_S \parallel_P o'$ in this case o' is put into a set U_i of the *unresolved* objects and o_S remains the selected object (line 5).
- $o' \prec_P o_S$ in this case o' is put into a set $D_{\succ}^{o_S}$, which contains the objects dominated by o_S according to $\precsim_P$, and o_S remains the selected object (line 6).
- $o_S \prec_P o'$ in this case S_i is added to the set $D_{\succ}^{o'} = D_{\succ}^{o'} \cup S_i$, which contains the objects dominated by o' according to $\precsim_P$ and o' becomes the selected object (i.e., $o_S = o'$, $S_i = \{o'\}$) (line 7).

[9] Recall that BNL after returning block B_i, if needed, runs over $R = R \setminus B_i$

After the algorithm completes the database scan there is no object among those processed that dominate objects in S_i. So objects in S_i are put into the block B_i in which Best collects the objects to be returned as the output of the i^{th}-step. However, there might be some objects o' in U_i dominated by o_S. For this reason the algorithm also compares the selected object with the objects in U_i (lines 9,10). At that point, if U_i is not empty, Best repeats the whole procedure but this time using U_i as input (i.e., another scan at the end of which a new set S_i will be inserted in B_i) (line 12). When at the end of the a scan, U_i gets empty the i^{th}-step is concluded and Best returns the next block B_i (line 13). If the number of the returned objects is more than k, then the algorithm stops. Otherwise, Best is executed again this time using as input only the objects in the sets $D_{\succ}^{o}$ for each $o \in B_i$ (i.e., $Input = \bigcup \{D_{\succ}^{o} \mid o \in B_i\}$) (line 15).

Best

input: a relation R, a preference expression P, an integer k

output: the $<_{\exists\forall}$-aware top-k objects of R according to P

1: $i = 0$, Result $= \varnothing$, $Input = R$

2: Repeat

3: Let as o_S the first $Active(o) \in Input$ // o_S is the selected object

4: While $\exists$ active object $o' \in Input$ not compared with o_S:

5: If $o_S \|_P o'$ then $U_i = U_i \cup o'$

6: ElseIf $o' \prec_P o_S$ then $D_{\succ}^{o_S} = D_{\succ}^{o_S} \cup o'$

7: ElseIf $o_S \prec_P o'$ then $D_{\succ}^{o'} = D_{\succ}^{o'} \cup S_i$, o_S=o' , $S_i = \{o'\}$

8: ElseIf $o' \sim_P o_S$ then $S_i = S_i \cup o'$

9: While $\exists o' \in U_i$ not compared with o_S :

10: If $o' \prec_P o_S$ then $D_{\succ}^{o_S} = D_{\succ}^{o_S} \cup o'$, remove o' from U_i

11: $B_i = B_i \cup S_i, S_i = \varnothing$

12: If $U_i \neq \varnothing$ then $Input = U_i$

13: Else *return* B_i; $|result| = |result| + |B_i|$

14: If |result|<k then

15: $Input = \bigcup \{D_{\succ}^{o} \mid o \in B_i\}$

16: $i = i + 1$

17: Else break

18: *Until* false

Figure 9: Best Algorithm

Best inherits the advantage (i.e., easy implementation) and the disadvantages (i.e., at least one dominance test for every active object in R, not progressive) from BNL. However, Best requires only one scan of the relation R independently to the number k of returned objects.

Best case time complexity: Like BNL, the best case time complexity of Best is $O(n)$ where n is the number of objects in R and occurs when the result (i.e., the top-k objects) is small comparing to n.

Worst case time complexity: Worst case time complexity of Best is $O(n^2)$ and occurs when all objects of R are incomparable to each other.

Space Complexity: The space complexity of Best is $O(|R|)$ where $|R|$ is the size of the input relation R (in pages), since the entire relation might be kept in the $D_{\succ}^{o}$ sets.

3.2 Query Based Ordering

3.2.1 Lattice Based Algorithm (LBA)

The main intuition behind this algorithm is that each block B_i of the answer w.r.t. a preference P corresponds to the result of a selection query QB_i (i.e., $B_i = ans(QB_i)$), neglecting object ordering. Block queries QB_i may be collected by "scanning" the active Cartesian Product $V(P, A)$ in a top-down manner, without having to calculate and store the latter. These queries are essentially unions of conjunctions of atomic selection conditions, containing all attributes that the user preference involves. LBA incrementally constructs and evaluates those queries starting from the one that returns the most preferred objects of R i.e., from QB_0, until the number of the returned objects exceeds k. For each query QB_i, $ans(QB_i)$ comprises incomparable (with respect to $\leq$) equivalence classes of objects and a query QB_i precedes QB_i' (i.e., $QB_i' < QB_i$) if between $ans(QB_i)$ and $ans(QB_i')$ it holds: $ans(QB_i') \leq_{\forall\exists} ans(QB_i)$. As a result, the retrieved objects are already ordered so there is no need to further compare them.

Given a preference $P = (dom(A), \precsim_P)$, each domain value $v_i \in V(P_i, A_i)$ and each tuple $v \in V(P, A)$ belong to an equivalence class declared by the symmetric parts $\sim_{Pi}$, $\sim_P$ of $\precsim_{Pi}$ and $\precsim_P$ respectively. To simplify the presentation of the algorithm in the rest of this chapter when we refer to domain values and tuples we shall actually mean their corresponding equivalence classes.

As we have seen in the previous chapter, a preference expression P over a set of attributes $A = \{A_1 \ldots A_m\}$, defines a preference relation (i.e., a non-antisymmetric partial preorder) over the elements $(v_1, v_2, \ldots, v_m)$ of the active preference domain $V(P, A)$. These elements essentially represent conjunctive

queries of the form $A_1 = v_1 \wedge A_2 = v_2 \wedge \ldots \wedge A_m = v_m$ which when executed will retrieve the matching objects. We call the respective ordering of queries the *Query Lattice*[10].

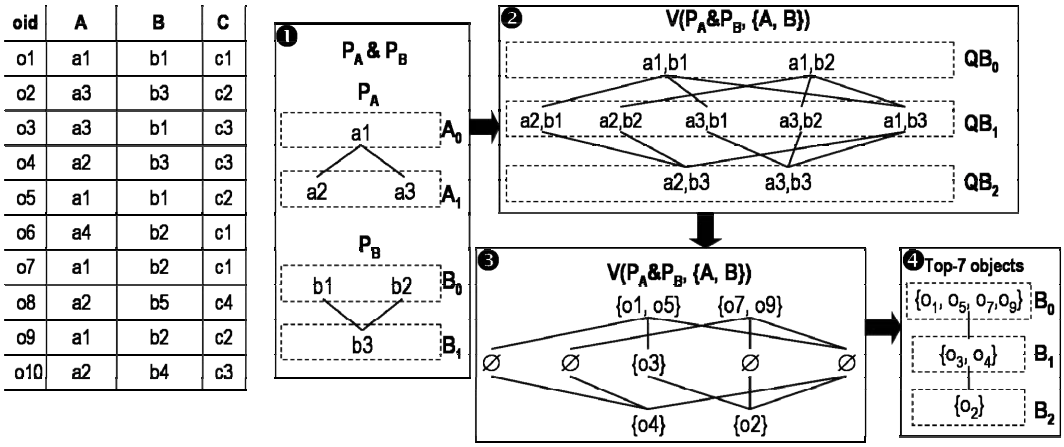

Figure 10: Query Ordering Framework

Consider, for example, the preference expression $P = P_A \,\&\, P_B$ of Figure 10.1, such that $a_2 \prec_{P_A} a_1$, $a_3 \prec_{P_A} a_1$, $b_3 \prec_{P_B} b_1$, $b_3 \prec_{P_B} b_2$. Figure 10.2 illustrates the Hasse diagram of $V(P_A \,\&\, P_B, \{A,B\})$; it also depicts the induced $<_{\forall\exists}$ block ordering $QB_2 <_{\forall\exists} QB_1 <_{\forall\exists} QB_0$ on $V(P_A \,\&\, P_B, \{A,B\})$. Clearly, to compute the most preferred objects (i.e., the top block B_0) w.r.t. $P = P_A \,\&\, P_B$, we need to execute the queries $A = a_1 \wedge B = b_1$ and $A = a_1 \wedge B = b_2$ deriving from the first query block QB_0. As both queries have non-empty results ($\{o_1, o_5\}$ and $\{o_7, o_9\}$, respectively, see Figure 10.3), we guarantee that they and only they return the most preferred objects (see Figure 10.4).

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 B_1. As we can see in Figure 10.3, from the five queries of the second lattice block QB_1, only $A = a_3 \wedge B = b_1$ has a non-empty result ($\{o_3\}$) which belongs to the next block of the anwser B_1. Yet, all other objects that belong to B_1, if any, have to result from queries that are successors[11] of the empty queries in QB_1,

[10] For simplicity, we omit a "true" top query and a "false" bottom query.

[11] Or, recursively their successors, in case they are empty.

and at the same time, are not successors of any other non-empty query in QB_1. This is the case of $A = a_2 \wedge B = b_3$ in QB_2, with result $\{o_4\}$, being child of the empty query $A = a_2 \wedge B = b_1$ and, at the same time, incomparable to the non-empty query $A = a_3 \wedge B = b_1$ of QB_1. On the contrary, $A = a_3 \wedge B = b_3$ in QB_2, although it is a child of two empty queries in QB_1, it is also a child of the non-empty one $A = a_3 \wedge B = b_1$ of QB_1; thus, its answer does not qualify for B_1. Recursively, we can compute the bottom block B_2 as illustrated in Figure 10.4.

As we already state, LBA aims to compute the $<_{\overline{\forall\exists}}$ block ordering of the top-k objects without actually needing to construct the induced ordering of objects. This is essentially achieved by exploiting the semantics of a preference expression and, in particular, by linearizing the active Cartesian Product $V(P, A)$ of all attribute values appearing in the expression. Going one step further, we don't even need to construct in advance and then linearize $V(P, A)$. Instead, we can simply construct its $<_{\overline{\forall\exists}}$ block ordering from the $<_{\overline{\forall\exists}}$ block ordering of its constituent atomic preferences. For example, in Figure 10.1 the $<_{\overline{\forall\exists}}$ block ordering of P_A is $A_1 = \{a_2, a_3\} <_{\overline{\forall\exists}} A_0 = \{a_1\}$ and of P_B is $B_1 = \{b_3\} <_{\overline{\forall\exists}} B_0 = \{b_1, b_2\}$. Thus, we introduce the following two theorems which provide the means to compute the $<_{\overline{\forall\exists}}$ block ordering of an arbitrary preference expression progressively.

Theorem 3.1:: Given the $<_{\overline{\forall\exists}}$ block orderings $X_{n-1} <_{\overline{\forall\exists}} \ldots <_{\overline{\forall\exists}} X_1 <_{\overline{\forall\exists}} X_0$, and $Y_{m-1} <_{\overline{\forall\exists}} \ldots <_{\overline{\forall\exists}} Y_1 <_{\overline{\forall\exists}} Y_0$ of two preferences P_X and P_Y, the $<_{\overline{\forall\exists}}$ block ordering $Z_{n+m-2} <_{\overline{\forall\exists}} \ldots <_{\overline{\forall\exists}} Z_1 <_{\overline{\forall\exists}} Z_0$ of preference $P = P_X \,\&\, P_Y$, will consist of $n+m-1$ blocks; each block Z_p will comprise elements only from blocks X_q and Y_r, such that $q + r = p$.

Proof 3.1: We start with the second part of the theorem, and use induction: It is obvious that the top (bottom) block Z_0 (say, Z_b, respectively) will derive from

the composition of the respective top (bottom) blocks X_0 (X_{n-1}) and Y_0 (X_{m-1}). The second block Z_1 must contain items which are worse than those of Z_0 in exactly one of their two constituents, i.e., worse either in X or in Y, but not in both. Furthermore, it should be worse by a distance of exactly 1 block in this constituent. To prove this, assume any item in Z deriving from constituents which either are worse in *both* X and Y, or are worse by a distance of *more than 1* blocks in X or Y; then, in both cases, such an item is obviously worse than some item(s) of Z_1, and thus it ought to belong in a block lower than Z_1. As 1+0=1 and 0+1=1 (for X, Y, and Z indices, respectively), the second part of the theorem holds for block Z_1, i.e., for a non trivial induction basis. For the induction hypothesis, assume that the theorem holds for block Z_k, i.e., $q+r=k$, for those X_q's and Y_r's which are the constituents for X and Y, respectively. Taking the induction step, it is obvious, by the previous discussion, that block Z_{k+1} should comprise those items originating either from X_{q+1} and Y_r, or from X_q and Y_{r+1} (i.e., the items from either of the precisely next blocks in X or Y, but not from both of those simultaneously); then, the new sum of the constituent blocks will have risen by exactly 1 to $k+1$; *q.e.d.* Using this result, and enumerating the values from 0 to $n-1$ and from 0 to $m-1$ we arrive at the actual number of Z-blocks, which is exactly $n+m-1$; and this completes the proof of the theorem.

Given the two block orderings $A_1 <_{\overline{\forall\exists}} A_0$ and $B_1 <_{\overline{\forall\exists}} B_0$ of Figure 10.1 for P_A and P_B respectively, the block ordering of preference $P = P_X \,\&\, P_Y$, will consist of 3 (i.e., 2+2-1) blocks. As we can see in Figure 10.2 the top block (QB_0) will be formed by combining elements from blocks whose sum of indices is 0, i.e., A_0 with B_0, the second (QB_0), from blocks whose sum of indices is 1, i.e., A_0 with B_1, and A_1 with B_0 , and the third (QB_0), from blocks whose sum of indices is 2,i.e., A_1 with B_1. The following theorem can be similarly proved:

Theorem 3.2: Given the $<_{\overline{\forall\exists}}$ block orderings $X_{n-1} <_{\overline{\forall\exists}} \ldots <_{\overline{\forall\exists}} X_1 <_{\overline{\forall\exists}} X_0$, and $Y_{m-1} <_{\overline{\forall\exists}} \ldots <_{\overline{\forall\exists}} Y_1 <_{\overline{\forall\exists}} Y_0$ of two preferences P_X and P_Y, the $<_{\overline{\forall\exists}}$ block ordering $Z_{n+m-2} <_{\overline{\forall\exists}} \ldots <_{\overline{\forall\exists}} Z_1 <_{\overline{\forall\exists}} Z_0$ of preference $P = P_X \triangleright P_Y$, will consist of $n \times m$ blocks; each block Z_p will comprise elements only from blocks X_q and Y_r, and it will hold $p = q \times m + r$. 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,…, X_0Y_{m-1}, X_1Y_0, …, $X_{n-1}Y_{m-1}$.

Algorithm LBA takes as input a relation R and a preference expression P involving a subset A of R's attributes. Then, it outputs progressively the $<_{\overline{\forall\exists}}$-aware top-$k$ objects of R. To this end, LBA relies on a internal representation of the sequence of blocks of the active Cartesian Product $V(P, A)$ (see Figure 10.2). In particular, array QB is used to hold in main memory only the structure of the $<_{\overline{\forall\exists}}$ block ordering of $V(P, A)$. The corresponding Query Lattice is not materialized but rather the queries needed to construct the requested blocks 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 $V(P_i, A_i)$ forming a block of $V(P, A)$. Going back to Figure 10, QB_0 contains the single element list $<0,0>$, standing for A_0, B_0, whereas QB_1 contains the list $<0,1> \rightarrow <1,0>$, standing for A_0, B_1 and A_1, B_0, respectively. The entire QB array of our example can be seen in Figure 11.

QB_0 : $<0,0>$
QB_1 : $<0,1> \rightarrow <1,0>$
QB_2 : $<1,1>$

Figure 11: The QB array of $P = P_1 \,\&\, P_2$

After computing QB (line 1), LBA iteratively calls *GetBlockQueries* (line 4) to create a list of associated conjunctive queries and *Evaluate* (line 5) in order to output successive blocks of objects until the top-k objects were retrieved (or $V(P, A)$ is exhausted).

LBA

input: an object relation R, a preference expression P, an integer k

output: the $<_{\overline{\forall\exists}}$-aware top-$k$ objects of R according to P

1: $QB = ConstructQueryBlocks(P.root)$

2: $result = i = 0$

3: *Repeat*

4: $Uq_i = GetBlockQueries(QB[i])$

5: $result += Evaluate(Uq_i)$

6: $i += 1$

7: *Until* $result \geq k$ *or* $i = |QB|$

Figure 12: *LBA* Algorithm

ConstructQueryBlocks returns the structure of the final expression result in the form of blocks. It traverses recursively a preference expression tree P (starting 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 $<_{\overline{\forall\exists}}$ block ordering. When & (line 6) and ▷ (line 7) appear as a preference relation between expressions $P.left$ and $P.right$, it calls *ParetoComp* or PriorComp to construct the corresponding QB. For leaves (i.e., for atomic preferences), their respective QB entries are computed (line 2) by PrefBlocks which for an atomic preference P_i derives its $<_{\overline{\forall\exists}}$ block ordering of $V(P_i, A_i)$. For example, in its "bottom left" recursion step *ConstructQueryBlocks* creates a QB with two entries QB_0: $<0>$ and QB_1: $<1>$ for the $<_{\overline{\forall\exists}}$ block ordering $A_1 <_{\overline{\forall\exists}} A_0$ of preference P_A.

ConstructQueryBlocks

input: a preference expression P

output: the QB array of P

1: If P is a leaf then // P is an atomic preference

2: $QB = \text{PrefBlocks}(V(P_i, A_i))$

3: Else

4: $QB\_left = ConstructQueryBlocks(P.left)$

5: $QB\_right = ConstructQueryBlocks(P.right)$

6: If $P.type = \text{"\&"}$ then $ParetoComp(P.left, P.right)$

7: Else PriorComp$(P.left, P.right)$

8: Return $P.QB$

Figure 13: *ConstructQueryBlocks* function

ParetoComp and PriorComp implement theorems 3.1 and 3.2 respectively. In particular, *ParetoComp* given two input preferences P_1 and P_2 computes the QB, for the case of $P_1 \& P_2$. As explained in theorem 3.1 QB will comprise $|P1.QB| + |P2.QB| - 1$ blocks; and the sum of the indices of each element in every block will equal the index of that block. On the other hand, PriorComp for its input preferences P_1, P_2 computes the QB for the case of $P_1 \triangleright P_2$. As defined in theorem 3.2, QB will comprise $|P1.QB| \times |P2.QB|$ blocks, and the order of the blocks follows the lexicographical order of the indices of the corresponding blocks.

ParetoComp

input: two operand nodes P_1 and P_2

output: the QB of node $P = P_1 \,\&\, P_2$

1: $|P.QB| = |P1.QB| + |P2.QB| - 1$

2: For $w = 0$ to $|P.QB|$

3: $P.QB[w] = \bigcup \{P1.QB[i] \times P2.QB[j] \mid i + j = w\}$

4: Return $P.QB$

Figure 14: *ParetoComp* function

PriorComp

input: two operand nodes P_1 and P_2

output: the QB of node $P = P_1 \triangleright P_2$

1: $w = 0$

2: $|P.QB| = |P1.QB| \times |P2.QB|$

3: For $i = 0$ to $|P1.QB| - 1$

4: For $j = 0$ to $|P2.QB| - 1$

5: $P.QB[w] = P1.QB[i] \times P2.QB[j]$

6: $w+ = 1$

7: Return $P.QB$

Figure 15: PriorComp function

Function *Evaluate* executes each query q of its input set Uq_i. It keeps track of non-empty queries in SQs, so that they are executed only once. Also, for the object block B_i currently processed, it keeps track of non-empty queries in $CurSQs$ (line 4) and of empty ones in FQs (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 SQs (thus avoiding to execute twice a non-empty query), and not in $CurSQs$ (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).

Evaluate

input: a list of queries

output: the next block B_i

1: For each q in Uq_i

2: If q not in SQs then

3: If $ans(q) != \varnothing$ then

4: $CurSQs \cup = \{q\}$; $B_i \cup = ans(q)$

5: Else $FQs \cup = \{q\}$

6: Else $FQs \cup = \{q\}$

7: While $FQs != \varnothing$

8: For each q in FQs

9: $FQs \setminus = \{q\}$

10: $Q_1 = \{q_1 \mid q_1 = child(q)\}$

11: For each q in Q_1

12: If q not in SQs then

13: If not q in $succ(q')$ forall q' in $CurSQs$ then

14: If $ans(q) != \varnothing$ then

15: $CurSQs \cup = \{q\}$; $B_i \cup = ans(q)$

16: Else $FQs \cup = \{q\}$

17: Else $FQs \cup = \{q\}$

18: $SQs \cup = CurSQs$; $CurSQs = \varnothing$

19: *output* B_i ; *return* $|B_i|$

Figure 16: *Evaluate* function

Function $child()$ (line 10), returns the direct children of its input query q. There are several ways to implement $child()$. In our case the implementation of $child()$ was based on the following observation: We already know the list element $< l_1, l_2, \ldots, l_m >$ of QB_i from which q has originated. Clearly, the direct children of q must originate only from one or more list elements of QB_{i+1}. For each list element $< l_1', l_2', \ldots, l_m' >$ of QB_{i+1}, let *lastblock* be the set that contains each $j \in \{1...m\}$ for which l_j is the index to the last block of the corresponding $<_{\overline{\forall}\exists}$ block ordering of $V(P_j, A_j)$. From the list elements $< l_1', l_2', \ldots, l_m' >$ of QB_{i+1} only those that satisfy the following property may point to children of q:

- there exists $c \notin lastblock$ such that $l_c' = l_c + 1$
- $\forall k \notin lastblock - \{c\}$ it holds $l_k' = l_k$.

Having identified those list elements $< l_1', l_2', \ldots, l_m' >$ of QB_{i+1} which directly point to children of $q \coloneqq A_1 = v_1 \wedge A_2 = v_2 \wedge \ldots \wedge A_m = v_m$, the queries to be returned by the function are produced as follows:

For each $j \in \{1...m\}$ such that $l_j' \neq l_j$:

- If $l_j' = l_j + 1$ then replace v_j with the direct children of v_j in the corresponding $P_j = (dom(A_j), \precsim_{P_j})$
- If $l_j' = 0$ then replace v_j with the maximal values of $\precsim_{P_j}$ that are related to v_j

3.2.1.1 $<_{\not\exists\overline{\forall}}$-aware LBA

Consider now the case where the size of $V(P, A)$ is very large compared to the number of available (active) objects. As a result LBA will have a lot of fruitless fetching attempts (for most of the queries q it will hold $ans(q) = \varnothing$). This will lead to poor performance since the algorithm will continuously keep searching $V(P, A)$ for possible exclusive successors of q that will probably result empty

answers too. In such a scenario it would be reasonable to adopt a more "relaxed" linear order of blocks that however will not go against the intuition "most-preferred objects first" which is probably the most important constraint that each linear order of blocks should satisfy. Therefore we define the $<_{\not\exists\bar{\forall}}$ order of blocks as follows:

Definition 3.1: Let $B_i <_{\not\exists\bar{\forall}} B_j$, iff $\forall[o] \in B_i$, $\not\exists[o'] \in B_j$ such that $[o'] \leq [o]$.

Therefore in an $<_{\not\exists\bar{\forall}}$ adaptation of LBA, when for a query q holds $ans(q) = \varnothing$ there is no need to search for possible exclusive successors of q since the identification of the incomparable objects is not a strict requirement here. The only (but important) difference between the $<_{\bar{\forall}\exists}$ and the $<_{\not\exists\bar{\forall}}$ variation of LBA is that the *Evaluate* function, is only responsible for fetching objects and no further examination is required.

Evaluate

input: a list of queries

output: the next block B_i

1: For each q in Uq_i

2: $B_i \cup = ans(q)$

3: *output* B_i; *return* $|B_i|$

Figure 17: *Evaluate* function for the $<_{\not\exists\bar{\forall}}$ variation of LBA

It is worth noticing that the $<_{\not\exists\bar{\forall}}$ variation of the LBA algorithm can be also sensitive to scenarios where the size of $V(P, A)$ is large compared to the number of available objects. Howerer due to the fact that the identification of incomparable objects is not a strict requirement, it is expected to be more efficient than the $<_{\bar{\forall}\exists}$ variation.

Moreover, since in the $<_{\not\exists\bar{\forall}}$ variation of LBA we are not actually forced to identify which queries yield empty queries and which not, we could employ some different rewriting techniques in order to construct queries which are more efficient to evaluate. So far, given a block query QB_i, for each of its tuples

$v = (v_1, v_2, \ldots, v_m) \in QB_i$ one conjunctive query of the form $q_j := A_1 = v_1 \wedge A_2 = v_2 \wedge \ldots \wedge A_m = v_m$ was formulated and executed individually. QB_i was defined as the union of those conjunctive queries. From now on, we will refer to this rewriting as MQ (*Multiple Queries*). A second rewriting approach would be to define QB_i as the disjunction of the conjunctions of the atomic selection conditions that each tuple $v \in QB_i$ defines. For example assuming that QB_i contains two tuples $v = (v_1 \ldots v_m)$ and $v' = (v_1' \ldots v_m')$, QB_i will have the following form:

$$QB_i := (A_1 = v_1 \wedge A_2 = v_2 \wedge \ldots \wedge A_m = v_m) \vee (A_1 = v_1' \wedge A_2 = v_2' \wedge \ldots \wedge A_m = v_m')$$

We refer to this rewriting as $SQ_{\vee}$ (*Disjunctive Single Query*). Finally, QB_i can be defined as the conjunctive query of m disjunctions (one for each attribute). Each disjunction refers to a specific attribute A_i and consists of every atomic selection condition that each tuple $v \in QB_i$ defines and refers to A_i where any repeated conditions are removed. For example assuming that QB_i contains two tuples $v = (v_1 \ldots v_m)$ and $v' = (v_1' \ldots v_m')$, QB_i will have the following form:

$$QB_i := (A_1 = v_1 \vee A_1 = v_1') \wedge (A_2 = v_2 \vee A_2 = v_2') \wedge \ldots \wedge (A_m = v_m \vee A_m = v_m')$$

We refer to this rewriting as $SQ_{\wedge}$ (*Conjunctive Single Query*).

3.2.1.2 Analytical Evaluation

In this section we analyze the complexity of LBA by focusing on the cost of computing the top block of the top-k objects. This choice is motivated by the fact that generating the top block has the same cost in the worst case as constructing the entire block ordering. Furthermore, it provides a common ground for comparison with existing algorithms evaluating skyline queries. LBA algorithm has a very small startup cost for constructing the $<_{\exists\forall}$ block ordering of the input atomic preferences $O(|V(P_i, A_i)|^2)$ and in general can be neglected. 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 *oids* and then fetching the matching objects from the disk. When (unclustered) B+-trees are used, 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 block ordering, the LBA cost is $O(r\times(\log|R|+|ans(q)|))$.

Best case time complexity: In the best case, only one query is required to construct B_0 and the number of returned objects is very small (especially for uniform data distributions). In particular, when $|Act(P,A)|>>|V(P,A)|$, the practical cost of LBA drops to $O(\log|R|)$.

Worst case time complexity: 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 objects (especially for skewed data distributions). Thus, the total cost of the index traversals will rise to $O(|V(P,A)|\times\log|R|)$ where $|V(P,A)|=\overset{m}{\underset{i=1}{\times}}|V(P_i,A_i)|$, while the I/O cost of their non-empty answers will be $O(|Act(P,A)|)$, bringing the total worst case cost up to $O(|V(P,A)|\times\log|R|+|Act(P,A)|)$. In particular, when $|Act(P,A)|<<|V(P,A)|$ and given that $\log|R|$ is usually small (3 to 6, depending on B+-tree[12] fan-out), the practical complexity of LBA in the worst case becomes $O(|V(P,A)|)$. It should be stressed that the above cost also characterizes the worst case LBA complexity when requesting only B_0.

Space Complexity: The space complexity of LBA depends on the size of the QB which will store in overall $\overset{m}{\underset{i=1}{\times}}\#blocks(P_i,A_i)$ list elements $<l_1,l_2,\ldots,l_m>$, where $\#blocks(P_i,A_i)$ is the number of blocks in the corresponding $<_{\overline{\forall\exists}}$ order of $V(P_i,A_i)$.

[12] Alternatively, hash indices could be used with a typical cost of 1-2 I/Os.

The (best, worst) time complexity of the $<_{\not\exists\forall}$ variation of LBA that follows the MQ approach for constructing queries is the same as the $<_{\exists\forall}$ variation. However in practise the $<_{\exists\forall}$ variation of LBA will evaluate fewer queries since the identification of incomparable objects is not a strict requirement in a $<_{\not\exists\forall}$ ordering. In case where one of the remaining two approaches (i.e., $SQ_{\vee}$ and $SQ_{\wedge}$) are followed, the algorithm will evaluate at most $|QB|$ queries where $|QB|$ is the size of the corresponding QB array of a preference expression P. Each of these queries will cost $O(|Act(P,A)|)$ in the worst case. So the overall compexity is now $O(|QB| \times |Act(P,A)|)$.

It is clear for someone to see that the performance of LBA is very sensitive to $\frac{|V(P,A)|}{|Act(P,A)|}$ ratio (where $Act(P,A)$ denotes the active objects of R w.r.t. P). If $\frac{|V(P,A)|}{|Act(P,A)|} < 1$ then almost for each query q it will hold $ans(q) \neq \varnothing$ and as a result only a relatively small number of queries needs to be evaluated in order to retreive the top-k objects. On the other hand if $\frac{|V(P,A)|}{|Act(P,A)|} > 1$ then for most of the queries it will hold $ans(q) = \varnothing$ and this will lead LBA to evaluate a large number of queries (in the worst case $|V(P,A)|$).

In LBA variations the retrieval of objects is performed in an ordered manner so there is no need to perform dominance tests to compare the already retreived objects. Furthermore assuming that available indexes exist, LBA algorithms will access only the objects that will be returned as the top-k objects and only once. Moreover it is LBA algorithms are progressive (i.e., they return the next block of the answer without having to previously compute the following blocks). However LBA algorithms are sensitive in scenarios where the size of $V(P,A)$ is very large compared to the number of available active objects (i.e., $\frac{|V(P,A)|}{|Act(P,A)|} >> 1$).

3.2.2 Threshold Based Algorithm (TBA)

When $V(P,A) >> Act(P,A)$, LBA will be forced to execute a big number of queries which yield empty answers, before succeeding to arrive at one with a non empty result. For this reason, we devise a second algorithm, called TBA, which is a hybrid of the Query Lattice presented previously and the dominance testing approaches ([29], [30]). TBA incrementally constructs and evaluates queries to quickly locate and fetch a small portion of R that includes the top-k objects. Unlike LBA, these queries are disjunctions of atomic selection conditions over just one attribute. In order to determine when the fetching of objects should stop TBA uses appropriate thresholds. These thresholds ensure that objects that were not fetched are worst than the ones that were already fetched (i.e., work as an upper bound of the unseen objects). For defining the ordering of queries, TBA takes into account the selectivities of the atomic selection conditions so that to avoid fetching more objects than those actually required. However, TBA needs to perform dominance tests for the already retrieved objects. Therefore it can be said that TBA adopts ideas from both query and object based approaches since it uses the specified user preferences to define an ordering over queries, howerer it also perfoms dominance tests for the retrieved objects.

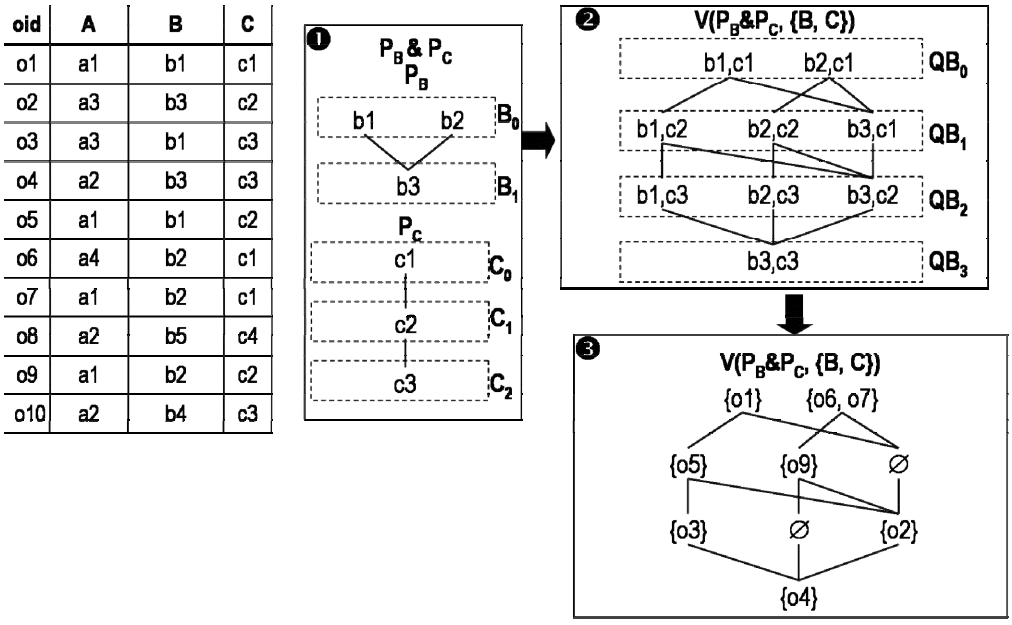

Figure 18: A Query Ordering framework example

Before we fully describe TBA lets see the intuition behind this algorithm. Assume, for example, the preference expression $P = P_B \,\&\, P_C$ of Figure 18, such that $c_2 \prec_{P_C} c_1, c_3 \prec_{P_C} c_2$ and $b_3 \prec_{P_B} b_2$, $b_3 \prec_{P_B} b_1$. The Hasse diagram of $V(P_B \,\&\, P_C, \{B, C\})$ and the induced $<_{\forall\exists}$ block ordering $QB_3 <_{\forall\exists} QB_2 <_{\forall\exists} QB_1 <_{\forall\exists} QB_0$ is is depicted in Figure 18.2. Like before, the top block QB_0 contains the maximal values of the active preference domain, since it combines elements from the top blocks B_0 and C_0 of the constituent atomic preferences P_B and P_C. It is clear that the corresponding value pairs on B or C behave as *thresholds*. For instance, there cannot be any object not inspected yet in the result, that has better values than $(b1, c1)$ and $(b2, c1)$.

Let us now consider, a disjunctive query q on attribute C formed by all active values of C_0; in our example, q is $C = c_1$ as there is only one value in C_0. Clearly, any object of R that does not belong to the result of q, cannot be better than objects matching pairs of values obtained by the next block C_1 of $V(P_C, C)$, i.e., the value pairs $B0 \times C1 = \{(b1, c2), (b2, c2)\}$. In other words, we lower the threshold by going one block "down" in $V(P_C, C)$ (i.e., the active terms of the attribute we chose to issue the disjunctive query q) while we keep the previous block for $V(P_B, B)$. Next, we need to check for dominance among the objects returned by q (e.g., $o1, o6, o7$); as we derive $o6 \sim_P o7$ and $o1 \|_P o6$, all three objects are undominated. Due to transitivity, if for each of the new threshold values in $B_0 \times C_1$ there is a more preferred object in the set of undominated objects of $ans(q)$, the latter actually constitutes the first block of the answer, i.e. the undominated objects of the whole relation. Repeating the process we can construct the block sequence of objects as depicted in Figure 18.

In the general case let preference $P = (dom(A_1) \times \ldots \times dom(A_m), \precsim_P)$ and assume that for each one of the atomic preferences $P_j = (dom(A_j), \precsim_{Pj})$ exists a $<_{\forall\exists}$

block order $X_j^{\,w} <_{\overline{\forall\exists}} \ldots <_{\overline{\forall\exists}} X_j^{\,1} <_{\overline{\forall\exists}} X_j^{\,0}$ of $Active_{Pj}(dom(A_j))/\sim_{Pj}$ where $X_j^{\,k}$ denotes the k^{th} block in the ordering.

Definition 3.2: We define as *threshold values* the Cartesian Product $Thres = X_1^{p1} \times X_2^{p2} \times \ldots \times X_m^{pm}$ where p_i is an index that refers to X_i and indicates the first block of X_i that was not fetched (initially each p_i points to $X_i^{\,0}$).

These values ensure that objects that were not fetched are worst than the ones that were already fetched (i.e., work as an upper bound of the unseen objects). It is worth noticing here, that *Thres* is a set of values of $V(P, A)$ contrary to quantitative threshold based algorithms ([2], [11], [20]) where thresholds are actually arithmetic scores. Clearly at any point in time, an object that was not already been fetched cannot be more preferred than a value in $Thres = X_1^{p1} \times X_2^{p2} \times \ldots \times X_n^{pn}$. More precisely the following theorem holds:

Clearly at any point in time, an object that was not already been fetched cannot be more preferred than a value in $Thres = X_1^{p1} \times X_2^{p2} \times \ldots \times X_n^{pn}$. More precisely the following theorem holds:

Theorem 3.3: For each active object o that was not been fetched there is a treshold value $t \in Thres$ such that $o \precsim_P t$.

Proof 3.3: Assume that there is an unseen (i.e., not fetched) active object o for which $o[A] = v$ where $v = (v_1 \ldots v_m)$ such that $\forall t = (v_1' \ldots v_m') \in Thres$ it holds $t \prec_P o$ or $t \|_P o$. Thus in that case tuple v must contain at least one atomic value v_i s.t $v_i' \prec_{Pi} v_i$ or $v_i' \|_{Pi} v_i$ where v_i' is the corresponding value of t for attribute A_i. But each v_i' is a maximal value of $V(P_i, A_i)$ that was not already been fetched, therefore for each v_i, v_i' it holds $v_i \precsim_{Pi} v_i'$.

In the rest of this section we will detail TBA (Figure 19). TBA starts (line 2) by

calling PrefBlocks that computes for each consistent atomic preference P_j, the $<_{\overline{\exists\forall}}$ block ordering over $V(P_i, A_i)$. The result is maintained internally by an array PB of lists whose elements hold only the block indices of the active terms of $V(P_i, A_i)$. The threshold values are stored in an array $Thres$ of size m (i.e., the total number of attributes A), and initially comprise the top blocks of all PB lists (line 3). Throughout its execution, TBA keeps in memory two sets with the objects that were fetched, but not yet returned: Dominated contains all objects for which some better were found, while Undominated contains the equivalence classes of objects for which no better object was met. Both sets are initially empty (line 4). Then, the following 4 steps are repeated, until the requested answer size is reached or $Act(P, A)$ is exhausted (line 12):

- TBA identifies the block of attribute A_i with the lowest selectivity (for all active values it contains), among those referred by $Thres$ (line 6) and the respective disjunctive query is executed.
- Function $OrderObjects$ is called (line 8) to pair-wise compare the returned objects and update Dominated, Undominated sets accordingly.
- $Thres$ is updated by obtaining the next best block of $V(P_i, A_i)$ (line 10).
- Function $GetNextBlock$ is called (line 11) next; depending on its input parameters it will output one or more blocks of the answer, and also update accordingly sets Dominated and Undominated.

Let us return to the termination case of exhausting $Act(P, A)$ before k is reached. This will happen when one of the lists in $Thres$ is exhausted (line 12). We prove this by reduction to the absurd: Assume that the list for attribute preference P_k is exhausted and yet there is an active object o with value $(v_1, \ldots, v_{k-1}, v_k, v_{k+1}, \ldots)$. Object o should contain active values on every attribute, so v_k should be active. Thus, v_k should have already been inspected, or else belong in the remaining part of P_k's list. Both cases contradict the hypothesis. This condition is treated trough a special value $bottom$, denoting the lowest of thresholds; using the bottom threshold as input, $GetNextBlock$

(lines 13-14) will find any set of undominated objects better than it, and thus will output the next blocks as required.

Threshold Based Algorithm

input: an object relation R, a preference expression P, an integer k

output: the $<_{\overline{\forall\exists}}$-aware top-$k$ objects of R according to P

1: *For* $j = 1$ *to* m //*for each atomic preference* P_i

2: $PB[j] = PrefBlocks(V(P_j, A_j))$

3: $Thres[j] = head(PB[j])$

4: Undominated=Dominated=$\varnothing$; $|result| = 0$

5: *Repeat*

6: $i = \min\_selectivity(Thres)$

7: $Q = \vee(A_i = v_j), \forall v_j \in Thres[i]$

8: *OrderObjects*(Ans(Q),Undominated,Dominated)

9: If next($PB[i]$) then

10: $Thres[i]$ = next($PB[i]$)

11: *GetNextBlock* (Undominated,Dominated))

12: Else

13: $Thres = \{bottom\}$

14: *GetNextBlock* (Undominated,Dominated))

15: *break*

16: Until $|result| >= k$

Figure 19: Threshold Based Algorithm

Function *OrderObjects* takes as input two sets of objects, *Input* and *Dom*, as well as a set of equivalence classes of objects *Und*. If empty, *Und* is initially filled with the class of the first object of *Input* (line 2). *OrderObjects* updates the sets *Dom* and *Und* after comparing every object o of *Input* against a single representative o' of all classes of objects in *Und*. Four cases may occur:

- If o is found worse than some o' (line 7), it is appended to *Dom* and it does not have to be compared against the rest of *Und*.

- If o is found equally preferred to some o' (line 10), it is appended to the class of o' in *Und* and again no more comparisons against the rest of *Und* are needed.

- If o is found better than some o' (line 11), the (flattened) class of o' is moved from *Und* to *Dom*; *OrderObjects* continues testing o with the rest of *Und*.

- If o is incomparable to o', comparisons continue with the rest of *Und*, without any further action. At the end of comparisons, if o is found not to be dominated by any *Und* element (line 12), a new class containing o is appended to *Und*.

OrderObjects

---

input: sets of objects *Input*, *Dom*, set of classes of objects *Und*

output: a pair of sets *UptDom*, *UptUnd*

1: $UptDom = Dom$

2: If $Und = \varnothing$ then $UptUnd = [o_1]$ // o_1 is the first active object of *Input*

3: Else $UptUnd = Und$

4: For each active object o in *Input*

5: IsDominated = false

6: For each o' in *UptUnd*

7: If $o \prec_P o'$ then

8: IsDominated = true

9: $UptDom \cup = \{o\}$; *break* //inner for

10: ElseIf $o' \sim_P o$ then $[o'] \cup = o$; *break*

11: ElseIf $o' \prec_P o$ then $UptUnd \setminus = [o']$; $UptDom \cup = \{o'\}$

12: If not(Dominated) then $UptUnd \cup = [o]$

13: return *UptDom*, *UptUnd*

Figure 20: *OrderObjects* function

Function *GetNextBlock* takes as input a set of dominated objects (*Dom*) and a set of undominated classes of objects (*Und*). Using the current threshold values

(*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. *GetNextBlock* checks whether for each of the threshold values in *Thres* there is a more preferred object in the set of undominated objects of *Und* (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 classes of objects is reset (lines 3-4). If k is not reached (line 5), *OrderObjects* is employed to partition the objects of *UptDom* in undominated and dominated ones (lines 6-7). With the sets updated in the previous step, *GetNextBlock* will be recursively applied (line 8), until either of the conditions in lines 2 or 5 fail.

GetNextBlock

input: sets of objects *Dom*, set of classes of objects *Und*

output: a pair of sets *UptDom*, *UptUnd*

1: $UptDom = Dom$; $UptUnd = Und$

2: If $(\forall t \in Thres, \exists o \in UptUnd \; s.t. \; t \prec_P o$ then

3: $B_i = UptUnd$; output B_i

4: $UptUnd = \varnothing$; $|result| += |B_i|$

5: If $|result| < k$ then

6: $Temp = UptDom$; $UptDom = \varnothing$

7: *OrderObjects*(*Temp*, *UptDom*, *UptUnd*)

8: *GetNextBlock*(*UptDom*, *UptUnd*)

Figure 21: *GetNextBlock* function

Similar to LBA, we can easily define a $<_{\not\exists\forall}$ variation of the TBA algorithm in cases we want a more "relaxed" linear ordering of blocks. The only difference between the $<_{\forall\exists}$ and the $<_{\not\exists\forall}$ variation of TBA is to "relax" the conditions in line 2 of *GetNextBlock* as follows: if $(\forall t \in Thres, \not\exists o \in UptUnd \; s.t. \; o \prec_P t)$

3.2.2.1 Analytical Evaluation

Similar to LBA, TBA has an initialization phase cost which comprises the block ordering of the involved preferences; the latter is a memory cost of $O(|V(P_i, A_i)|^2)$ and in general can be neglected. The cost of TBA is mainly due to the number of disjunctive queries it has to execute in order to retrieve the top-k objects. Assuming that there are available indices in each attribute that the preference involves, a disjunctive query over one attribute is usually evaluated by traversing the available index on the involved attribute, computing the union of the oids and then fetching the matching objects from the disk. When (unclustered) B^+-trees are used, 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 compute the top-k objects, TBA's cost is $O(r \times (\log|R| + |ans(q)|))$. However, queries involve now only disjunctions of preference terms per attribute while the returned objects are not exclusively active but may include inactive ones matching at least one active attribute term. In addition, the fetched objects are compared pair-wise.

Best case time complexity: 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., ideally k). Thus, the cost of pair-wise dominance testing can be neglected. In particular, when $|Act(P, A)| >> |V(P, A)|$ the best case practical cost of TBA is $O(\log(|R|))$.

Worst case time complexity: 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 objects. The total number of queries executed in this case is given by the number of blocks of preference terms per attribute $\sum_{i=1}^{m} \#blocks(P_i, A_i)$. Assuming a factor c [13] of extra inactive objects fetched w.r.t.

[13] Recall that TBA uses the most selective attribute terms and thus the number of inactive tuples expected to be fetched is relatively small.

the number of active ones, in the worst case TBA cost is $O(\sum_{i=1}^{m} \#blocks(P_i, A_i) \times \log(|R|) + c \times |Act(P, A)|)$ for I/Os and $O(|Act(P, A)|^2)$ for main memory objects comparisons. In particular, when $|Act(P, A)| >> \sum_{i=1}^{m} \#blocks(P_i, A_i)$, the practical complexity of LBA in the worst case becomes $O(|Act(P, A)|^2)$.

Space Complexity: The space complexity of TBA is $O(|R|)$, since the entire relation might be fetched and stored into $\text{Undominated}, \text{Dominated}$ sets.

TBA exploits selectivities of the atomic selection conditions so that to avoid fetching more objects than those actually required. Moreover TBA algorithm is progressive and thus suitable for on-line processing. However, TBA will access not only the top-k objects but also a portion of the active and inactive ones and probably more than once. Finally TBA needs to perform dominance tests for the retrieved objects. Compared to LBA, TBA is more sensitive to the number of active objects (due to dominance tests), and, at the same time, much less affected by the size of $V(P, A)$ (i.e., sum vs. product of the of active preference domains sizes $V(P_i, A_i)$). This is one of the subjects of our experiments reported in the following chapter.

Chapter 4: Experimental Evaluation

In this chapter we experimentally evaluate the top-k algorithms presented in Chapter 3. The goal of this evaluation is to measure the performance as well as the sensitivity of the presented algorithms against realistic data distributions and sizes of preferences. Specifically, following the methodology widely used in the literature ([5], [27], [29], [31]) we consider different kinds of synthetic databases (correlated and uncorrelated) exhibiting various value distributions. We also vary the number of the atomic preferences involved, the complexity of each atomic preference, the composition operators, and the databases size.

4.1 Experimental Environment

All our experiments are carried out on a Pentium 4 CPU at 2.66 GHz with 1 GB of main memory. The operating system is Windows XP Pro SP2. The benchmark databases and intermediate results are stored on a 20 GB hard disk. We opted for an open source, rather than a commercial, framework for the implementation of our work, thus, all algorithms were implemented in Java on top of the PostgreSQL 8.1 Query Engine. Each benchmark database follows the relation schema $R(A_1, A_2, \ldots, A_{10})$ where the domain of attributes is given respectively by the sets $dom(A_1) = \{a_1, a_2, \ldots, a_{20}\}, dom(A_2) = \{b_1, b_2, \ldots, b_{20}\}, \ldots, dom(A_{10}) = \{j_1, j_2, \ldots, j_{20}\}$. Each database tuple is 100 bytes long; all indexes were implemented as B^+ trees. In some experiments we also implemented hash indexes. Testing has shown no difference in performance, while the index size and build time for hash indexes was much worse. Therefore, the performance figures presented in the rest of this chapter employ B^+ trees.

4.2 Preference and Testbed Generator

Each atomic preference P_j is created by first defining the size and the number of blocks $\#blocks(P_j)$ of the poset $(dom(A_j), \precsim_{P_j})$[14]. Next all blocks are populated by randomly allocating all the nodes to them, at least one to each of them. Then the poset is formed by randomly connecting nodes so that each node of a block B_d can be linked only with nodes of block B_{d-1}. Block B_0 will contain the maximal elements of the poset. We study three different kinds of databases that differ in the distribution of values over attributes:

- **Uniform:** for this type of database, all attribute values are generated independently using a uniform distribution. Thus, all distinct values of a domain have the same selectivity.

- **Correlated:** for a given preference P**,** a correlated database represents a testbed in which objects which are good (with respect to P) in one attribute are also good in the other attributes too. We produce a random object in a correlated database as follows. First, all attribute values are generated using a uniform distribution. For each active object, if a maximal value appears in one attribute then the object is forced to receive maximal values in the other attributes too. Otherwise the object remains unchangeable. Therefore, in a correlated database a large portion of the available active objects are undominated according to P (i.e., belong to B_0).

- **Anti-Correlated:** for a given preference P, an anti-correlated database represents a testbed in which objects which are good in one attribute are bad in another attribute. We produce a random object in an anti-correlated database as follows. First, all attribute values again are generated using a uniform distribution. If an active object has a value in

[14] Of course the size should be larger than the number of blocks since each block must contain at least one node

an attribute that belongs to the top half blocks of the corresponding atomic preference it would randomly receive a value in another attribute that would belong in one of the bottom half blocks of the corresponding atomic preference and so on. Thus, the first blocks of $V(P, A)$ do not exist in an anti-correlated database.

We also studied testbeds that followed the exponential distribution $\beta e^{-x\beta}$ (mean $\beta^{-1} = 10$). In particular, for each attribute A_j we defined a list that contained all distinct values of $dom(A_j)$. The positioning of each value in the list was performed in several manners: randomly, optimistically (i.e., active values first), pessimistically (i.e., active values last). The first value in the list would appear $0{,}1 \times e^{\frac{-1}{10}} \times |R|$ times in the database, the second $0{,}1 \times e^{\frac{-2}{10}} \times |R|$ and so on. However, the results were similar to the three kinds of testbeds already described and as a result we only show the results for the uniform, correlated and anti-correlated testbeds. In our experimental presentations, unless stated otherwise, we ask for the top-1 (i.e., the undominated) objects accrording to $P = P_1 \,\&\, P_2 \rhd P_3$ which is our default preference where $P_1 = (dom(A_1), \precsim_{P_1})$, $P_2 = (dom(A_2), \precsim_{P_2})$, $P_3 = (dom(A_3), \precsim_{P_3})$. Figure 22 depicts their corresponding Hasse diagrams:

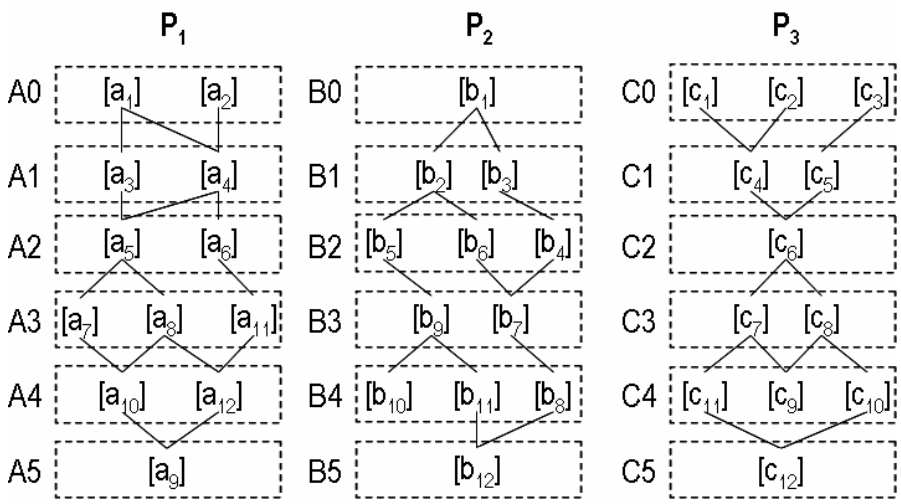

Figure 22: Hasse diagram for our default atomic preferences

4.3 Metrics

4.3.1 Experimental parameters

In order to analyse the results of our experiments, we define the following metrics[15]:

1. $\frac{|V(P,A)|}{|dom(A)|}$ ratio: If $\frac{|V(P,A)|}{|dom(A)|}=1$ then all objects of the database are active. This metric is affected obviously if we alter $|V(P,A)|$. Specifically we can decrease $\frac{|V(P,A)|}{|dom(A)|}$ by increasing the dimensionality of the preference expression and increase it by also increasing the atomic preferences size (i.e., increasing the number of active values in each domain).
2. $\frac{|Act(P,A)|}{|R|}$ ratio**:** the more this ratio is close to 1 the more dominance tests will be performed (for the algorithms that perform dominance tests). Again this metric is affected by varying the dimensionality of the preference expression and/or by changing the size of atomic preferences. In particular by increasing the dimensionality we decrease $\frac{|Act(P,A)|}{|R|}$ ratio while by increasing atomic preferences size we increase it.
3. $\frac{|q_{k,P}(R)|}{|Act(P,A)|}$ ratio**:** for a specific k, this ratio is actually the portion of active objects that are top-k. Clearly, we can alter metric 3 if we vary the number of requested objects k for each testbed.

[15] Recall that assuming a preference P over a non empty set of attributes A, $dom(A)$ and $V(P,A)$ are the Cartesian Products of domains and of active value sets respectively, $Act(P,A)$ is the set of active objects of R w.r.t P and $q_{k,P}(R)$ the set that contains the top-k objects.

4. $\frac{|V(P,A)|}{|Act(P,A)|}$ ratio**:** represents the Cartesian Product space in which active objects are distributed. If $\frac{|V(P,A)|}{|Act(P,A)|}<1$ then almost for each tuple $v \in V(P,A)$ there exists object $o \in R$ such that $o[A]=v$. We can decrease $\frac{|V(P,A)|}{|Act(P,A)|}$ ratio by increasing the size of the database and increase it by increasing the dimensionality of the preference.

4.3.2 Performance parameters

In order to present the major factors affecting the performance of each algorithm we also define the following metrics:

5. *Total* (*running*) *time* of each algorithm. *Total time* comprises into the *Database time* (i.e., the time needed by the DBMS to run the queries and to return the results) and the *Main memory time* (i.e., the time needed by each algorithm to run if all objects were available in memory).
6. $\frac{|Act(P,A)\_seen|}{|Act(P,A)|}$ ratio**:** Is the portion of the active objects that were processed (besides the ones that were returned as top-k) to the total number of active objects.
7. $\frac{|Inact(P,A)\_seen|}{|Inact(P,A)|}$ ratio**:** Is the portion of the inactive objects that were processed to the total number of inactive objects.
8. $|queries\_evaluated|$: is the number of queries that each algorithm evaluates in order to retrieve the top-k objects.
9. The *number of dominance tests* that each algorithm performs. Dominance test requires performing at most one "$\precsim$" test over each of the m attributes of the objects on which atomic preferences are expressed. If we assume that the cost of one subsumption check is that of reachability in graphs then its cost is $O(|E|)$ where E denotes the graph edges. Summarizing, the more atomic preferences we have, and the

more "better than" relations each atomic preference involves, the more expensive the dominance test becomes. Assuming that the top-k objects are partitioned into c classes of equivalence, then an algorithm will perform at least $\frac{c \times (c-1)}{2} + |Act(P,A)\_seen|$ dominance tests (i.e., the number of tests needed to compare the c representatives plus at least one test for each other active object that the algorithm sees). Of course, this holds for the algorithms that perform dominance tests.

Note that metrics 7 and 8 are meaningful only for TBA since for BNL and Best it holds $\frac{|Act(P,A)\_seen|}{|Act(P,A)|} = \frac{|Inact(P,A)\_seen|}{|Inact(P,A)|} = 1$ while for LBA it holds $\frac{|Act(P,A)\_seen|}{|Act(P,A)|} = \frac{|Inact(P,A)\_seen|}{|Inact(P,A)|} = 0$ regardless of the database or the preference expression that is used.

4.4 Query Patterns and Evaluation Plans

Beginning in release 8.1, PostgreSQL has the ability to combine multiple indexes (including multiple uses of the same index) to handle cases that cannot be implemented by single index scans. The system can form *AND* and *OR* conditions across several index scans. To combine multiple indexes, the system scans each needed index and prepares a *bitmap* in memory giving the locations of table rows that are reported as matching that index's conditions. The bitmaps are then *ANDed* and *ORed* together as needed by the query. Finally, the actual table rows are visited and returned. The table rows are visited in physical order, because that is how the bitmap is laid out; this means that any ordering of the original indexes is lost. Now we describe how actually PostgreSQL evaluates each query pattern that TBA and LBA produce.

- TBA constructs and evaluates queries which are simply disjunctions of atomic selection conditions over just one attribute. A general query q_i of the form following $q_i := A_j = v_1 \vee \ldots \vee A_j = v_m$, is broken down into m

separate scans of an index on A_j, each scan using one of the disjunctions. The results of these scans are then *ORed* together to produce the result.

- LBA that follows the MQ rewriting constructs a set of conjunctive queries q_j of the form $q_j := A_1 = v_1 \wedge A_2 = v_2 \wedge \ldots \wedge A_m = v_m$. For the implementation of q_j each index with the appropriate query clause is used and then the index results are *ANDed* together to identify the result rows.

- LBA that follows the $SQ_{\vee}$ rewriting constructs queries which are disjunctions of the conjunctions of the atomic selection conditions. First for a general query q_j of the form $q_j := (A_1 = v_1 \wedge A_2 = v_2 \wedge \ldots \wedge A_m = v_m) \vee (A_1 = v_1' \wedge A_2 = v_2' \wedge \ldots \wedge A_m = v_m')$ the result rows for each of the conjunctions are identified as described before (i.e., each index with the appropriate query clause is used and then the index results are *ANDed* together). Then the *ANDed* results are *ORed* together to produce the actual results of q_j.

- LBA that follows the $SQ_{\wedge}$ rewriting produces queries which are defined as conjunctive queries of m disjunctions (one for each attribute). An example of such a query could be the following: $q_j := (A_1 = v_1 \vee A_1 = v_1') \wedge (A_2 = v_2 \vee A_2 = v_2') \wedge \ldots \wedge (A_m = v_m \vee A_m = v_m')$. Now initially the result rows for each of the disjunctions are identified (i.e., each index with the appropriate query clause is used and then the index results are *ORed* together). Then the *ORed* results are *ANDed* together to produce the actuall results of q_j.

4.5 The effect of database size

In order to evaluate the effect of the database size on our techniques, we use our default preference $P = P_1 \,\&\, P_2 \triangleright P_3$ of Figure 22 and vary the cardinality of the database from 10 to 1000 MB for each kind of database (uniform, anti-correlated, correlated). It is easy for someone to see that since $V(P,A)$ remains fixed here we will have more and more active objects by increasing the size of the database due to possible duplicates values. In other words, the larger the database gets the smaller $\frac{|V(P,A)|}{|Act(P,A)|}$ ratio becomes. Also the size of $q_{k,P}(R)$ becomes larger. For these reasons, the number of dominance tests that each algorithm needs to perform increases too. The following tables illustrate the metrics for each of the three testbeds.

Metrics \ MB	10	50	100	500	1.000
$\frac{\lvert V(P,A)\rvert}{\lvert dom(A)\rvert}$	1.728 / 8.000 = 0.216	1.728 / 8.000 = 0.216	1.728 / 8.000 = 0.216	1.728 / 8.000 = 0.216	1.728 / 8.000 = 0.216
$\frac{\lvert Act(P,A)\rvert}{\lvert R\rvert}$	21.410 / 100.000 = 0.214	107.599 / 500.000 = 0.215	215.001 / 1.000.000 = 0.215	1.079.549 / 5.000.000 = 0.215	2.160.391 / 10.000.000 = 0.216
$\frac{\lvert q_{k,P}(R)\rvert}{\lvert Act(P,A)\rvert}$	69 / 21.410 = 0.003	382 / 107.599 = 0.003	754 / 215.001 = 0.003	3.829 / 1.079.549 = 0.003	7.500 / 2.160.391 = 0.003
$\frac{\lvert V(P,A)\rvert}{\lvert Act(P,A)\rvert}$	1.728 / 21.410 = 0.0810	1.728 / 107.599 = 0.0160	1.728 / 215.001 = 0.0080	1.728 / 1.079.549 = 0.0016	1.728 / 2.160.391 = 0.0007

Table 1: Metric values for the Uniform Testbed

Metrics \ MB	10	50	100	500	1.000
$\frac{\lvert V(P,A)\rvert}{\lvert dom(A)\rvert}$	1.728 / 8.000 = 0.216	1.728 / 8.000 = 0.216	1.728 / 8.000 = 0.216	1.728 / 8.000 = 0.216	1.728 / 8.000 = 0.216
$\frac{\lvert Act(P,A)\rvert}{\lvert R\rvert}$	21.410 / 100.000 = 0.214	107.599 / 500.000 = 0.215	215.001 / 1.000.000 = 0.215	1.079.549 / 5.000.000 = 0.215	2.160.391 / 10.000.000 = 0.216
$\frac{\lvert q_{k,P}(R)\rvert}{\lvert Act(P,A)\rvert}$	9.074 / 21.410 = 0.424	45.381 / 107.599 = 0.422	90.723 / 215.001 = 0.422	453.615 / 1.079.549 = 0.420	907.230 / 2.160.391 = 0.420
$\frac{\lvert V(P,A)\rvert}{\lvert Act(P,A)\rvert}$	1.728 / 21.410 = 0.0810	1.728 / 107.599 = 0.0160	1.728 / 215.001 = 0.0080	1.728 / 1.079.549 = 0.0016	1.728 / 2.160.391 = 0.0007

Table 2: Metric values for the Correlated Testbed

Metrics \ MB	10	50	100	500	1.000
$\frac{\lvert V(P,A)\rvert}{\lvert dom(A)\rvert}$	$\frac{1.728}{8.000}$ 0.216	$\frac{1.728}{8.000}$ 0.216	$\frac{1.728}{8.000}$ 0.216	$\frac{1.728}{8.000}$ 0.216	$\frac{1.728}{8.000}$ 0.216
$\frac{\lvert Act(P,A)\rvert}{\lvert R\rvert}$	$\frac{21.410}{100.000}$ 0.214	$\frac{107.599}{500.000}$ 0.215	$\frac{215.001}{1.000.000}$ 0.215	$\frac{1.079.549}{5.000.000}$ 0.215	$\frac{2.160.391}{10.000.000}$ 0.216
$\frac{\lvert q_{k,P}(R)\rvert}{\lvert Act(P,A)\rvert}$	$\frac{534}{21.410}$ 0.025	$\frac{2.815}{107.599}$ 0.026	$\frac{5.656}{215.001}$ 0.026	$\frac{28.283}{1.079.549}$ 0.026	$\frac{56.464}{2.160.391}$ 0.026
$\frac{\lvert V(P,A)\rvert}{\lvert Act(P,A)\rvert}$	$\frac{1.728}{21.410}$ 0.0810	$\frac{1.728}{107.599}$ 0.0160	$\frac{1.728}{215.001}$ 0.0080	$\frac{1.728}{1.079.549}$ 0.0016	$\frac{1.728}{2.160.391}$ 0.0007

Table 3: Metric values for the Anti-correlated Testbed

4.5.1 Uniform Testbed

Figures 23 and 24 illustrate respectively the total time and the number of dominance tests of the various algorithms in the uniform testbed. Figure 25 shows the total execution time of the algorithm (i.e., database plus main memory time), while figures 26 and 27 depict the scalability over the database size for the two proposed algorithms. Clearly, LBA outperforms all other algorithms by several orders of magnitude. For example for the 1000 MB testbed BNL takes almost 1.000 sec while LBA consumes only 7 sec which outperforms the former by 3 orders of magnitude. Due to the fact that the size of the database increases and $V(P,A)$ remains fixed, all tuples of $V(P,A)$ exist and as a result, for LBA the queries of the first Query Lattice block suffice for computing the answer (in our testbed we need to execute only |A0|×|B0|×|C0|=6 queries). The only effect in performance is that these queries are more expensive to evaluate since $q_{k,P}(R)$ increases and more objects need to be fetched the larger the database gets. Compared with other algorithms we observe that LBA not only has better performance but also is more scalable. TBA maintains a significant advantage over BNL and Best (1 order of magnitude) and the difference increases fast when the database becomes bigger. This is due to the fact that TBA will not require in this case any threshold renewal therefore it fetches and processes only a small portion of the database. In this specific experiment TBA fetched only the 5% of the database objects which includes almost 8% of active objects and only 4% of the inactive ones. The overall runtime for BNL, Best increased significantly since they need to process more data objects and perform more dominance tests. Thus, BNL and

Best are very sensitive to the size of the database. In particular, for databases larger than 100 MB, Best exhibites poorer performance compared to BNL. Since Best has more memory requirements, Java's garbage collector is forced to run more times which is time-consuming. Best could not terminate successfully for the 1000 MB database due to the prohibitive size of the algorithm's memory requirements.

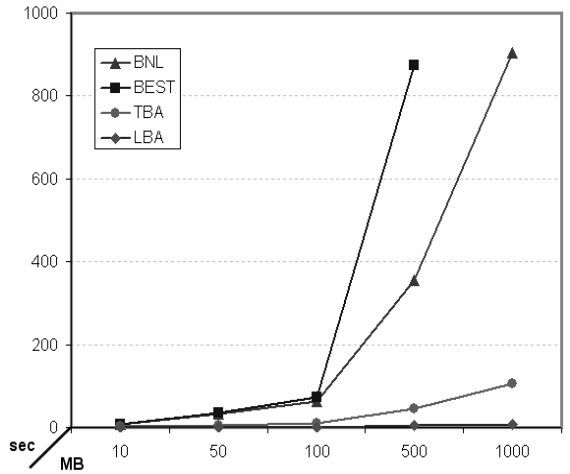

Figure 23: Total time

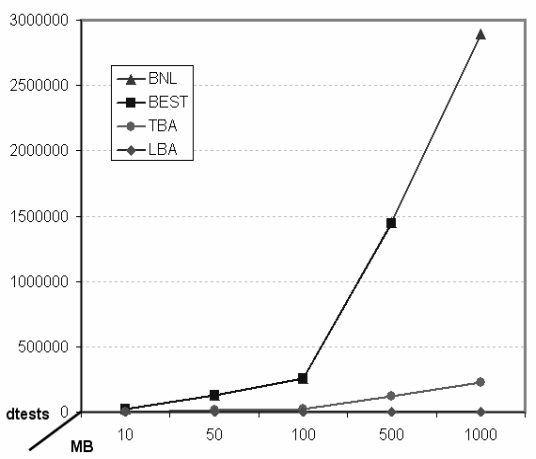

Figure 24: # dominance tests

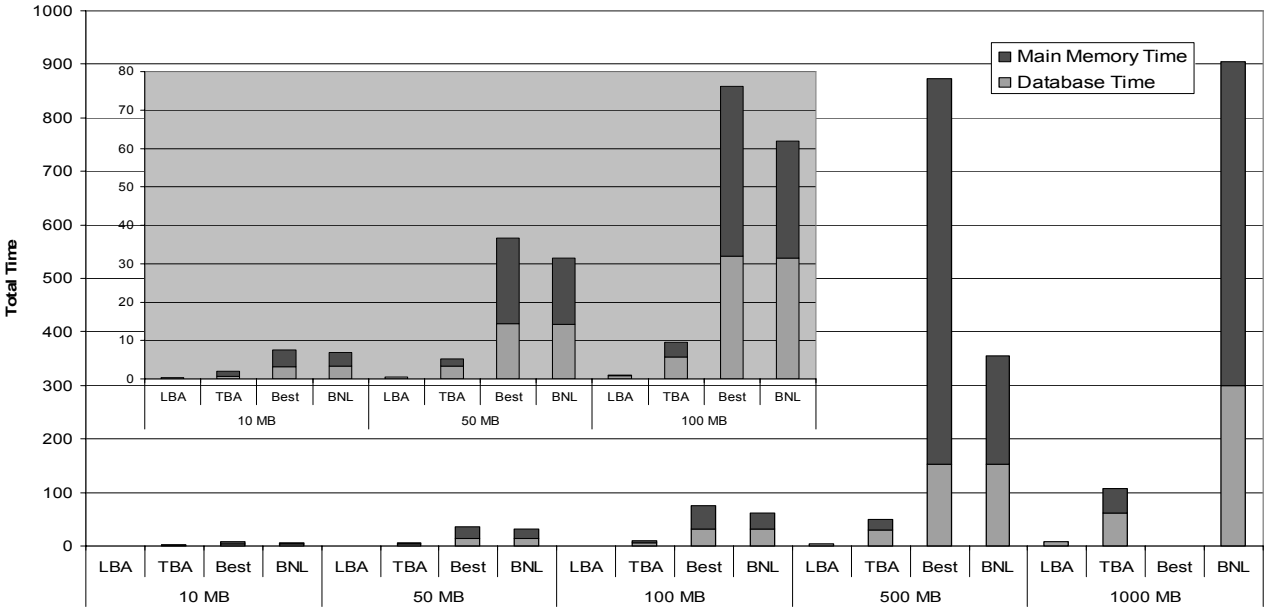

Figure 25: Total Time Analysis

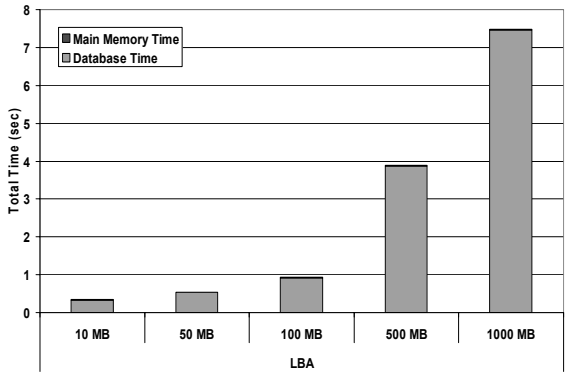

Figure 26: LBA scalability over database size

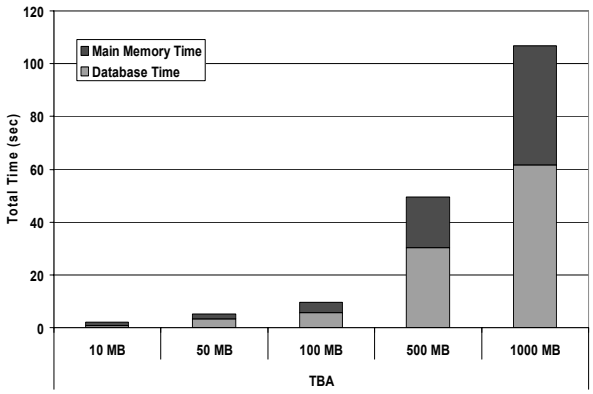

Figure 27: TBA scalability over database size

4.5.2 Correlated Testbed

Figures 28 to 32 show the performance of the various algorithms in the correlated testbed. The main characteristics of the correlated testbeds are that almost 40% of active objects are undominated objects and belong to the result. However, the number of equivalence classes in which the objects of the result are partitioned is the same as in the uniform testbeds. As we can see, the relative performance is unchanged compared to the uniform testbed. The only differences are that:

- Due to the nature of the correlated testbed, we have an increase of the answer size for each query issued by LBA and TBA, which are identical to those of the uniform testbed case.
- Moreover, the growth of the result itself causes a worth mentioning increase of the number of dominance tests for the respective algorithms (TBA, BNL, Best). This can be explained as follows: Assume that the top-k objects are partinioned into c classes of equivalence. Now let an incoming object o that belongs to the result which is equivalent to one of the c representatives and therefore incomparable to the remaining ones. Then in average $\frac{c}{2}$ dominance tests need to be performed in order to put o into the corresponding class. Now assume an object o' that does not belong to the result (i.e., o' is worst than some representatives). Then, in average less that $\frac{c}{2}$ dominance tests need to be performed to o' in order to find a representative that is better than o'. Hence, in average more dominance tests are performed for an object that belongs to the result compared to an object that does not. To conclude, in correlated testbeds the overall number of dominance tests increases because the size of the result is bigger than the corresponding result in uniform testbed.

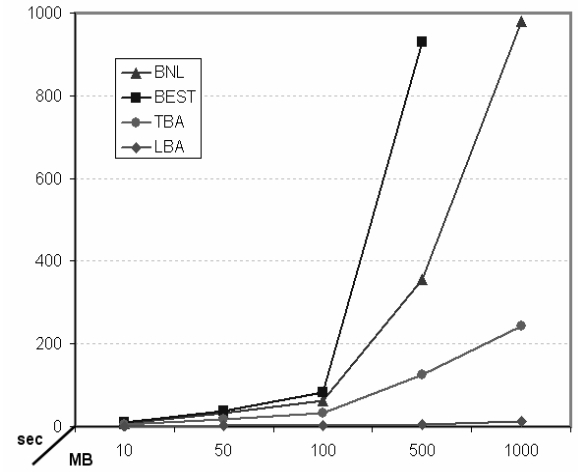

Figure 28: Total time

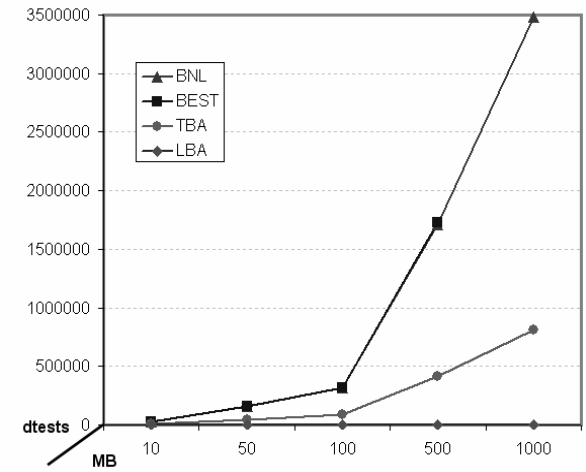

Figure 29: # dominance tests

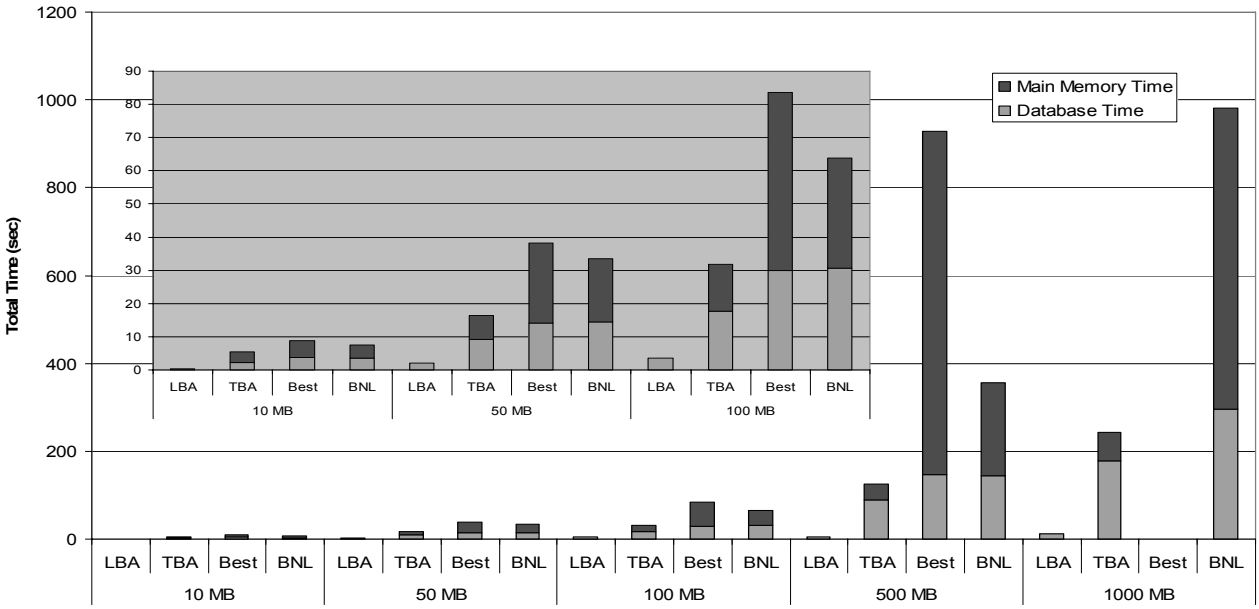

Figure 30: Total Time Analysis

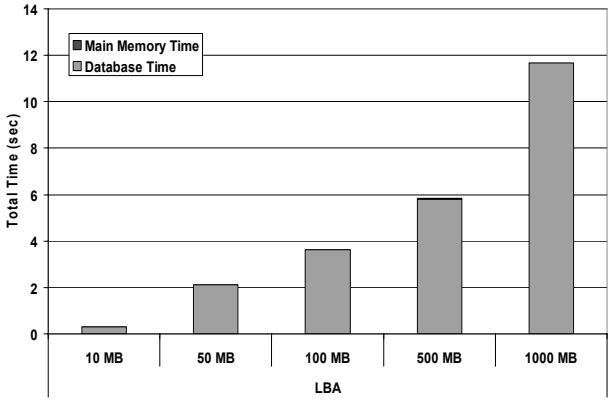

Figure 31: LBA scalability over database size

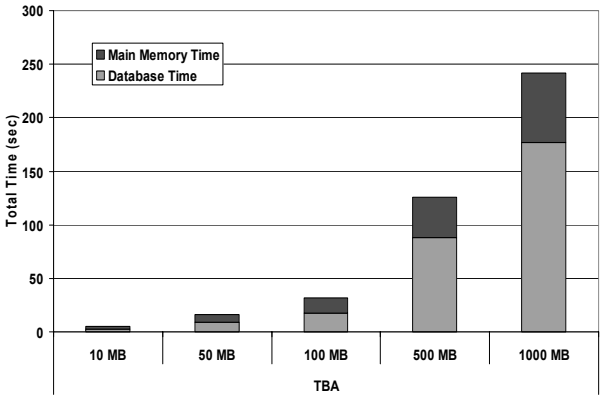

Figure 32: TBA scalability over database size

4.5.3 Anti-Correlated Testbed

In figures 33 to 37 we illustrate the performance of the various algorithms in the anti-correlated testbed. The relative performance is again unchanged compared to the uniform and the correlated testbeds, with only the following differences:

- LBA and TBA have an increased database time since both algorithms need to evaluate more queries in order to retrieve the top-k objects. In particular, in the anti-correlated testbeds LBA evaluates 550 and TBA 4 queries contrary to the uniform and the correlated testbeds where 6 and 1 queries need to be evaluated respectively.
- We have an increased number of dominance tests that the algorithms perform due to the facts that we have more objects in the result and because the top-k objects are partinioned into more equivelent classes compared to uniform testbeds. Specifically, in the uniform testbeds we have 6 classes of equivalence while in anticorrelated ones we have 40 classes.

TBA and BNL exhibit a similar behavior in the anticorrelated testbed and that is because TBA needs to fetch (and compare) a significant portion of the database. For example, in an anticorrelated testbed the percentage of active objects that TBA fetches increases almost to 60%. However, it is worth noticing that although TBA requires almost the same number of dominance test compared to BNL and Best there is a significant difference in their main memory processing time. This is due to the fact that the latter also includes the time needed by the algorithm to check if an object is active or not. To conclude, BNL and Best are penalized by the fact that they need to perform such checks for all objects of the database contrary to TBA. Above 500MB, Best was unable to terminate successfully.

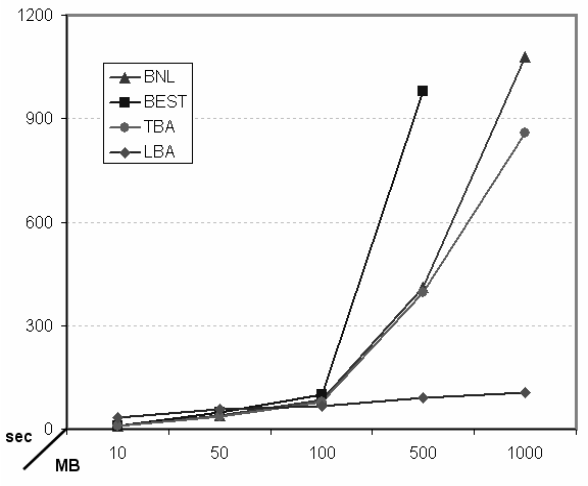

Figure 33: Total time

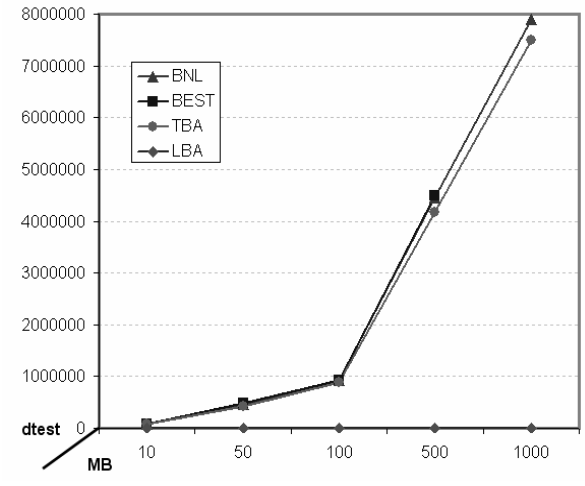

Figure 34: # dominance tests

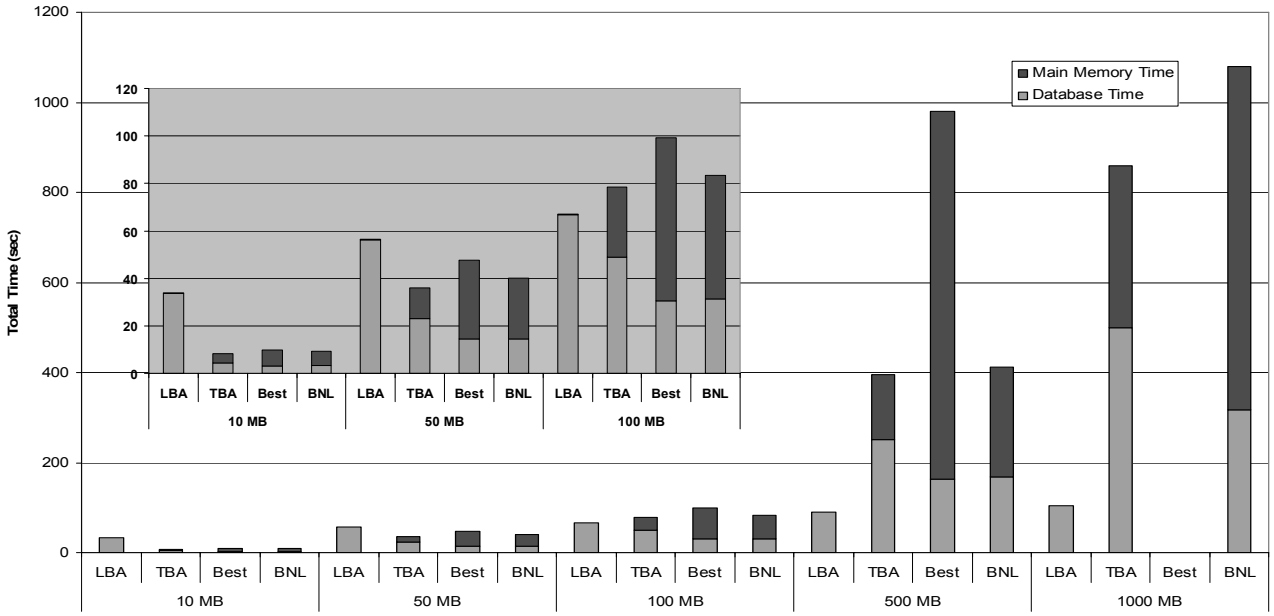

Figure 35: Total Time Analysis

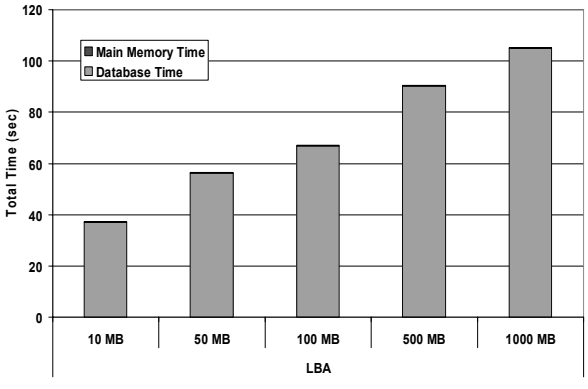

Figure 36: LBA scalability over database size

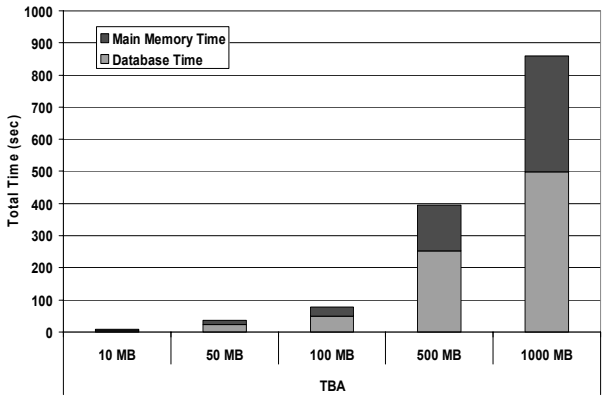

Figure 37: TBA scalability over database size

4.6 The effect of atomic preferences size

In order to study the effect of the atomic preference's size we used our default preference of Figure 22 and enhanced the size of each atomic preference $P_i = (dom(A_i), \precsim_{P_i})$ to involve more values from the corresponding domain $dom(A_i)$ until all values of $dom(A_i)$ to take part in $\precsim_{P_i}$ (in that case we will have $\frac{|V(P,A)|}{|dom(A)|} = \frac{|Act(P,A)|}{|R|} = 1$). So, increasing the size of each atomic preference has the effect of increasing the number of active objects and thus the number of dominance tests that need to performed. However, $\frac{|V(P,A)|}{|Act(P,A)|}$ ratio remains fixed since in the uniform testbed all values have the same selectivity. We initially increased the size of each atomic preference up to 16 and then up to 20. The number of blocks of each poset remained fixed. The enhancement was performed as follows: each additional node is randomly distributed between the blocks. Then the poset is reformed by randomly connecting the added nodes of a block B_d only with nodes of block B_{d-1}. In Table 4 we can see the metrics for this experiment in which we used a 100 MB uniform testbed.

Metrics \ Poset Size	12	16	20
$\frac{\lvert V(P,A)\rvert}{\lvert dom(A)\rvert}$	$\frac{1.728}{8.000} = 0.216$	$\frac{4.096}{8.000} = 0.512$	$\frac{8.000}{8.000} = 1.0$
$\frac{\lvert Act(P,A)\rvert}{\lvert R\rvert}$	$\frac{215.001}{1.000.000} = 0.215$	$\frac{511.434}{1.000.000} = 0.511$	$\frac{1.000.000}{1.000.000} = 1.0$
$\frac{\lvert q_{k,P}(R)\rvert}{\lvert Act(P,A)\rvert}$	$\frac{754}{215.001} = 0.003$	$\frac{3.019}{511.434} = 0.005$	$\frac{5.036}{1.000.000} = 0.005$
$\frac{\lvert V(P,A)\rvert}{\lvert Act(P,A)\rvert}$	$\frac{1.728}{215.001} = 0.008$	$\frac{4.096}{511.434} = 0.008$	$\frac{8.000}{1.000.000} = 0.008$

Table 4: Metric values (increasing atomic preference size)

Again the clear winner is LBA. In all instances of the experiment, LBA outperforms BNL and Best by 2 orders of magnitude. TBA maintains a significant advantage over BNL and the difference increases the larger the poset's size gets since TBA processes fewer active objects than BNL. The percentage of active objects that TBA fetches varies from 8% to almost 12%. BNL is significantly affected due to the need to perform more dominance tests.

Best could not terminate successfully when the size of the poset exceeds 16 due to the prohibitive size of the algorithm's memory requirements.

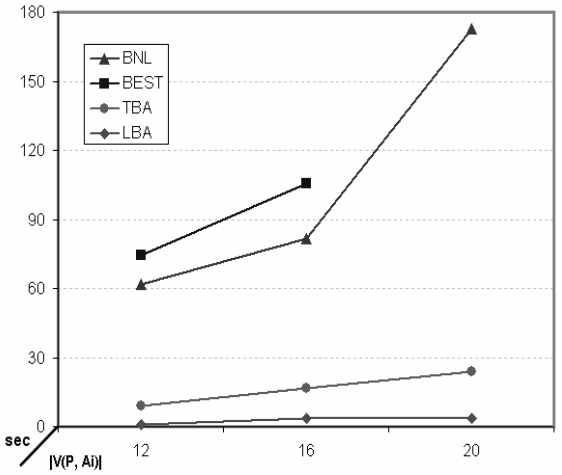

Figure 38: Total time

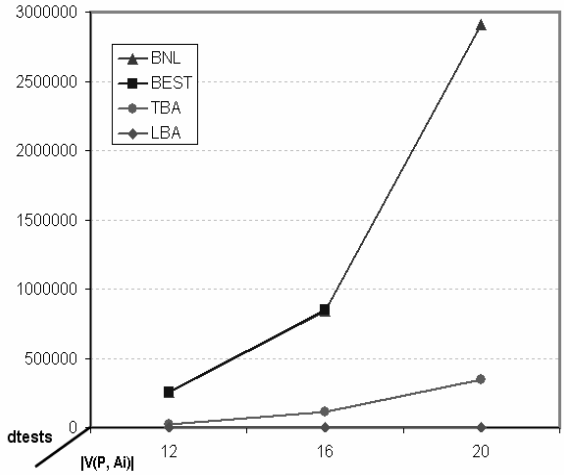

Figure 39: # dominance tests

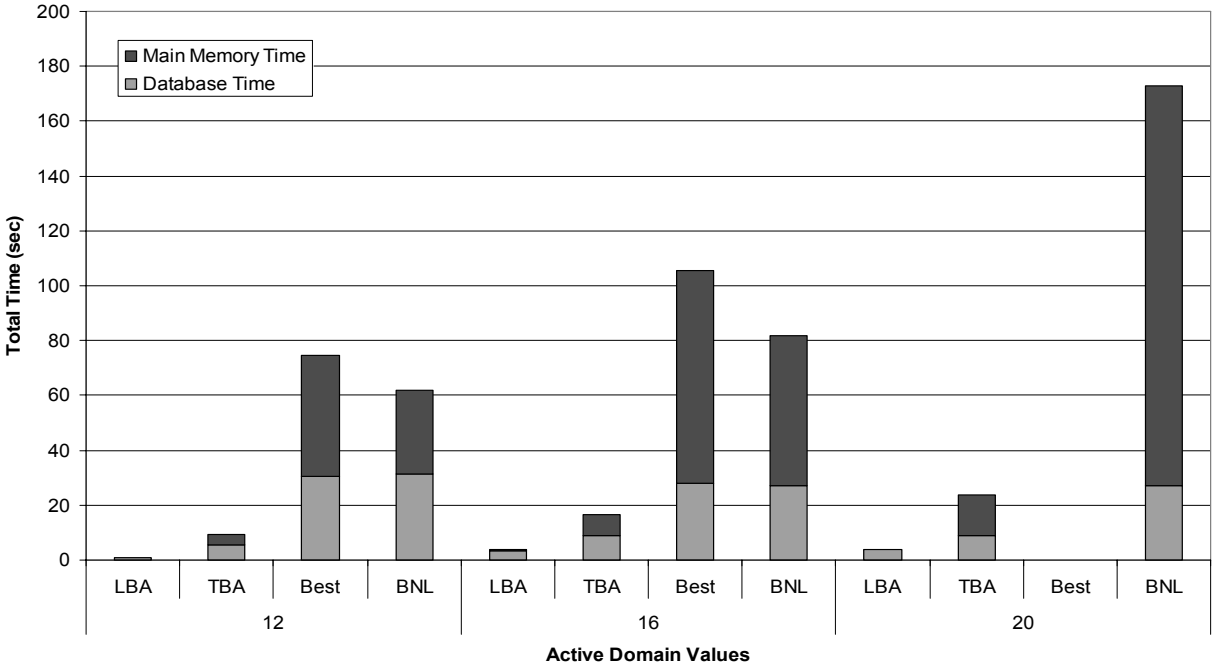

Figure 40: Total Time Analysis

No new blocks were added to the existing ones as the effect of increasing existing blocks' sizes is much stronger, both for LBA and TBA. For the former, this is due to the fact that the evaluation of a block B_i, engages the execution of all the queries in the respective QB_i block, thus, the more they are, the longer it will take. As for the latter, the "wider" a block is, the "longer" its selected disjunctive query will be and the bigger its answer size will get.

4.7 The effect of preference dimensions

In order to study the effect of dimensionality (i.e., the number of attributes involved in a preference expression), we used a 100 MB uniform testbed and varied the number of the atomic preferences m between 2 and 5. Each additional atomic preference was created as described in section 4.2. They have been composed using pareto and prioritized preferences. Clearly, regardless of the type of composition, the more attributes involved the more $|V(P,A)|$ increases while $|Act(P,A)|$ decreases. As a result, the larger m gets, metrics $\frac{|V(P,A)|}{|dom(A)|}$ and $\frac{|Act(P,A)|}{|R|}$ decrease while $\frac{|V(P,A)|}{|Act(P,A)|}$ ratio increases. In this particular experiment this happened when m changed from 4 to 5. Table 5 and Table 6 depict the metrics for this specific experiment.

Metrics \ dimensions	2	3	4	5
$\frac{\|V(P,A)\|}{\|dom(A)\|}$	$\frac{144}{400}$ 0.360	$\frac{1.728}{8.000}$ 0.216	$\frac{20.736}{160.000}$ 0.130	$\frac{248.832}{3.200.000}$ 0.078
$\frac{\|Act(P,A)\|}{\|R\|}$	$\frac{359.206}{1.000.000}$ 0.360	$\frac{215.001}{1.000.000}$ 0.215	$\frac{129.158}{1.000.000}$ 0.219	$\frac{77.453}{1.000.000}$ 0.077
$\frac{\|q_{k,P}(R)\|}{\|Act(P,A)\|}$	$\frac{4.954}{359.206}$ 0.014	$\frac{754}{215.001}$ 0.004	$\frac{44}{129.158}$ 0.0003	$\frac{131}{77.453}$ 0.002
$\frac{\|V(P,A)\|}{\|Act(P,A)\|}$	$\frac{144}{359.206}$ 0.0004	$\frac{1.728}{215.001}$ 0.008	$\frac{20.736}{129.158}$ 0.160	$\frac{248.832}{77.453}$ 3.213

Table 5: Metric values (increasing dimensionality-pareto composition)

Metrics \ dimensions	2	3	4	5
$\frac{\|V(P,A)\|}{\|dom(A)\|}$	$\frac{144}{400}$ 0.360	$\frac{1.728}{8.000}$ 0.216	$\frac{20.736}{160.000}$ 0.130	$\frac{248.832}{3.200.000}$ 0.078
$\frac{\|Act(P,A)\|}{\|R\|}$	$\frac{359.206}{1.000.000}$ 0.360	$\frac{215.001}{1.000.000}$ 0.215	$\frac{129.158}{1.000.000}$ 0.219	$\frac{77.453}{1.000.000}$ 0.077
$\frac{\|q_{k,P}(R)\|}{\|Act(P,A)\|}$	$\frac{4.954}{359.206}$ 0.014	$\frac{754}{215.001}$ 0.004	$\frac{44}{129.158}$ 0.0003	$\frac{12}{77.453}$ 0.0002
$\frac{\|V(P,A)\|}{\|Act(P,A)\|}$	$\frac{144}{359.206}$ 0.0004	$\frac{1.728}{215.001}$ 0.008	$\frac{20.736}{129.158}$ 0.160	$\frac{248.832}{77.453}$ 3.213

Table 6: Metric values (increasing dimensionality-prioritized composition)

As m increased, $|q_{k,P}(R)|$ decreased both in prioritized and in pareto composition. In the latter case, though, when $\frac{|V(P,A)|}{|Act(P,A)|}$ becomes larger than 1, $|q_{k,P}(R)|$ started increasing again. This behaviour is explicable if we follow the nature of the two operators. For the case of the prioritized composition, each time a new individual preference is added, the objects of the previous top block B_0, and only those, are candidates to belong to the new B_0', too. This is due to the "left to right" priority nature of the $\triangleright$ operator. So, the new top block B_0' will comprise $|B_0|$ or less objects.

For the case of the pareto composition, on the other hand, while $\frac{|V(P,A)|}{|Act(P,A)|}<1$, there are enough objects of $Act(P,A)$ to match the structure of $V(P,A)$, so $|q_{k,P}(R)|$ decreases with the $Act(P,A)$ decrease. But, when, eventually, it holds that $\frac{|V(P,A)|}{|Act(P,A)|}>1$, meaning that $Act(P,A)$ contains less and less objects while $V(P,A)$ grows wider, the probability of the former objects to be incomparable to each other rises, too. This leads to a high probability for each of these objects to belong to the new top block B_0, thus, increasing $|q_{k,P}(R)|$ as m increases again. Figure 41 and Figure 42 show the total times of the various algorithms as a function of dimensionality for pareto and prioritized composition respectively. LBA performs well until $\frac{|V(P,A)|}{|Act(P,A)|}$ becomes larger than 1 (i.e., when the preference contains more than 4 attributes). At that point, the degradation of LBA is caused by the need to evaluate a large number of empty queries (see Figure 43 and Figure 44) in order to search the large space of $V(P,A)$ where the top-k objects are distributed. TBA performs better than LBA when $\frac{|V(P,A)|}{|Act(P,A)|}>1$ and this is due to the fact that TBA needs to evaluate fewer queries than LBA. For example, for 5 attributes in a pareto preference LBA evaluates 772 queries while TBA only 6. This difference becomes more important as the number of attributes increases

and especially when the preference expression contains ▷ operators. TBA performs better in the prioritized composition and that is because a fetching attempt here drops threshold values more than the same fetching attempt in the pareto composition and that event leads to faster termination of the algorithm. In this experiment, since $|Act(P,A)|$ decreases and the size of the testbed remains fixed, the performance of BNL and Best mostly depends on the size of $q_{k,P}(R)$. When $q_{k,P}(R)$ decreases (e.g., in low dimensionality in the pareto composition or in prioritized composition), BNL and Best exhibit good scalability. On the other hand, when $q_{k,P}(R)$ increases (e.g., in high dimensional pareto composition) their performances drop since more pairwise comparisons are performed.

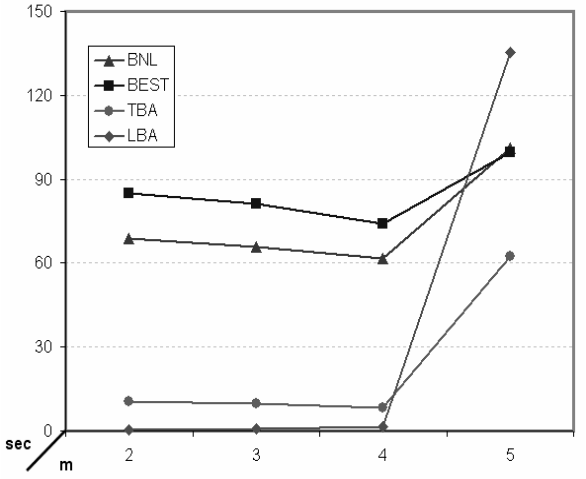

Figure 41: Total time, uniform testbed (100 MB)

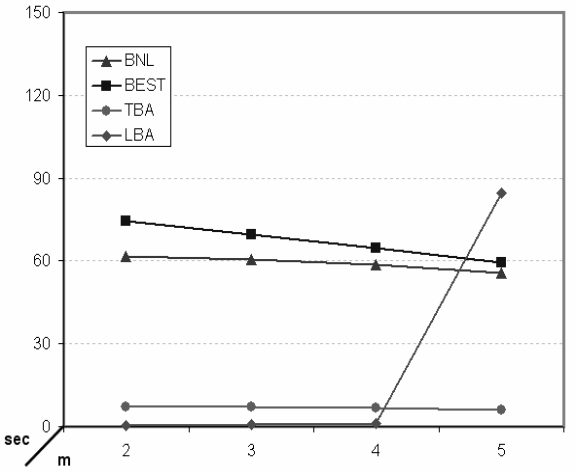

Figure 42: Total time, uniform testbed (100 MB)

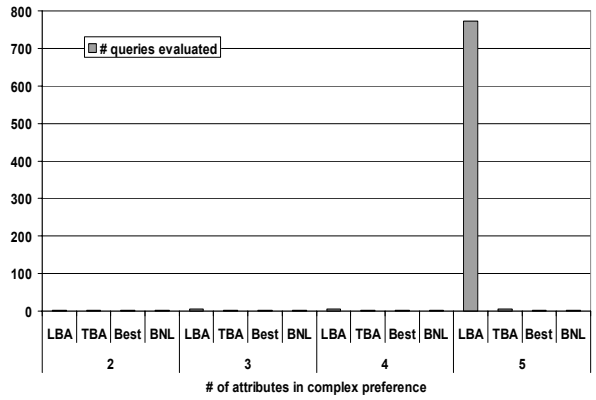

Figure 43:# queries evaluated - pareto composition

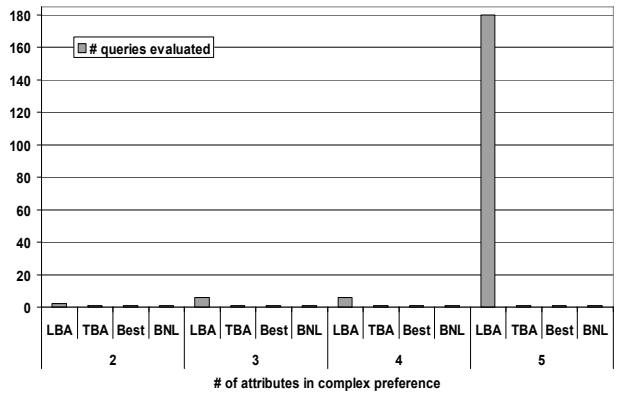

Figure 44: # queries evaluated-prioritized composition

4.8 Effect of the $<_{\not\exists\overline{\forall}}$ ordering

In this experiment we want to study the impact of adopting the more "relaxed" $<_{\not\exists\overline{\forall}}$ linear order of blocks vs. $<_{\overline{\forall\exists}}$ and identify possible performance trade-offs. Specifically we repeated experiment of section 4.7 (as dimensionality seems to be the most crucial factor in the performance of LBA and TBA) but this time we also included the $<_{\not\exists\overline{\forall}}$ variations of our algorithms. In particular, we included the following query rewritings:

- $LBA-MQ$: the $<_{\not\exists\overline{\forall}}$ variation of LBA that follows the MQ rewriting (i..e., one conjunctive query per tuple of $V(P,A)$).
- $LBA\text{-}SQ\text{-}conj$: the $<_{\not\exists\overline{\forall}}$ variation of LBA that follows the $SQ_{\wedge}$ rewriting (i.e., one conjunction of disjunctions per block of tuples of $V(P,A)$)
- $LBA\text{-}SQ\text{-}disj$: the $<_{\not\exists\overline{\forall}}$ variation of LBA that follows the $SQ_{\vee}$ rewriting (i.e., one disjunction of conjunctions per block of tuples of $V(P,A)$)
- $TBA-relaxed$: the $<_{\not\exists\overline{\forall}}$ variation of TBA

Figure 45 to 50 illustrate the performance of the various query rewritings in LBA and TBA algorithms with respect to the number of attributes for pareto and prioritized composition. As it was expected the $<_{\not\exists\overline{\forall}}$ variations of LBA, TBA are more efficient than the corresponding $<_{\overline{\forall\exists}}$ ones, since the identification of all of the incomparable objects is not a strict requirement in the $<_{\not\exists\overline{\forall}}$ order. Moreover, the LBA variations are more efficient than the TBA variation. $LBA\text{-}SQ\text{-}conj$ execution times are the best and outperforms all other $<_{\not\exists\overline{\forall}}$ variations by 1 order of magnitude. Although $LBA\text{-}SQ\text{-}conj$ in each case constructs the same number of queries as $LBA\text{-}SQ\text{-}disj$ does, the evaluation of $LBA\text{-}SQ\text{-}conj$'s queries needs fewer index scans and hence leads to better performance (see section 4.4). Note that we do not plot the results of

$TBA-relaxed$ in the prioritized composition because the algorithm behaves exactly like TBA.

We also replaced simple-key indexes with a complex-key one. In that case $LBA\text{-}SQ\text{-}disj$ had better performance than $LBA\text{-}SQ\text{-}conj$. However, the existence of a complex-key index in practical cases is rare since complex-key indexes are unlikely to be helpful unless the usage of the table is extremely stylized (e.g., when there are constraints on the leading-leftmost columns). Therefore we still propose $LBA\text{-}SQ\text{-}conj$ against $LBA\text{-}SQ\text{-}disj$.

Nevertheless there is a trade-off between the performance and the number of the top-k objects that the $<_{\not\exists\forall}$ variations of LBA, TBA actually return. For example in the 5 dimensional experiment of the pareto composition, TBA and LBA returned 131 objects while the $<_{\not\exists\forall}$ variations returned only 6. Similarly in the 5 dimensional experiment of the prioritized composition, TBA and LBA returned 12 objects while the $<_{\not\exists\forall}$ variations of the algorithms returned only 1. This is explained by the fact that the identification of the incomparable objects is not a strict requirement in the $<_{\not\exists\forall}$ order (recall that the $<_{\not\exists\forall}$ order does not go against the intuition "most-preferred objects first"). Conclusively, we can say that the $<_{\not\exists\forall}$ variations could be very useful in practical cases with preference expressions of high dimensionality. However, by paying the price of "sacrificing" a subset of top-k objects in terms of efficiency.

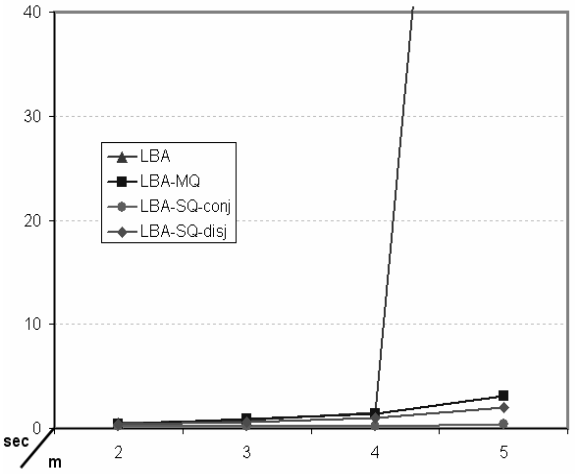

Figure 45: Total time, uniform testbed (100 MB)

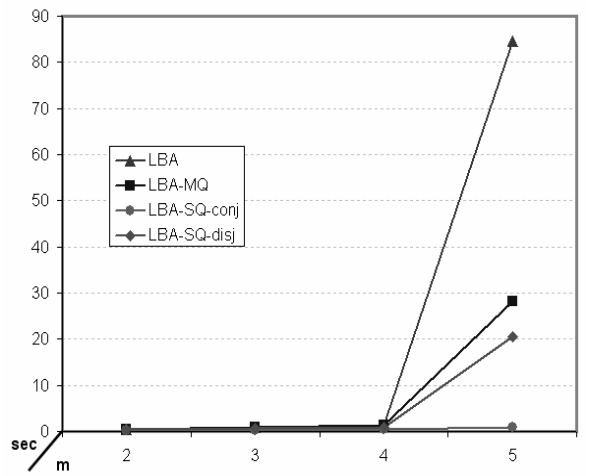

Figure 46: Total time, uniform testbed (100 MB)

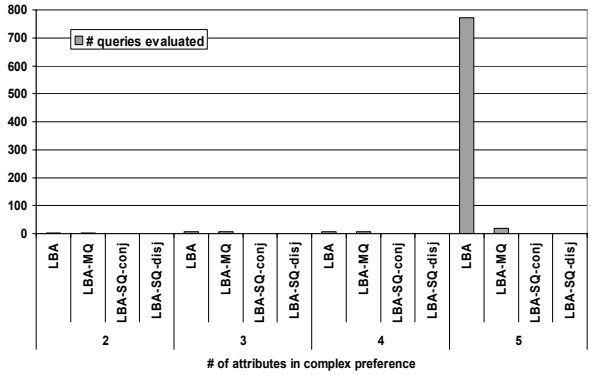

Figure 47: : # queries evaluated-pareto composition

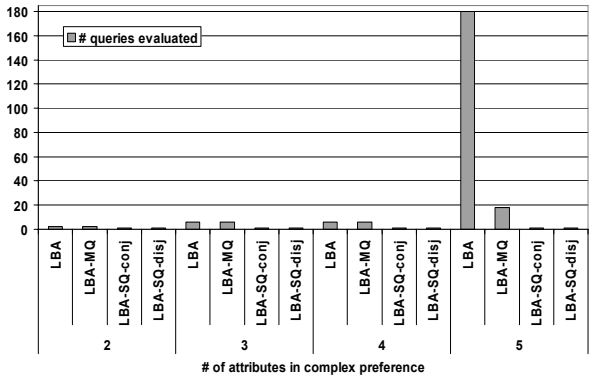

Figure 48: # queries evaluated-prioritized composition

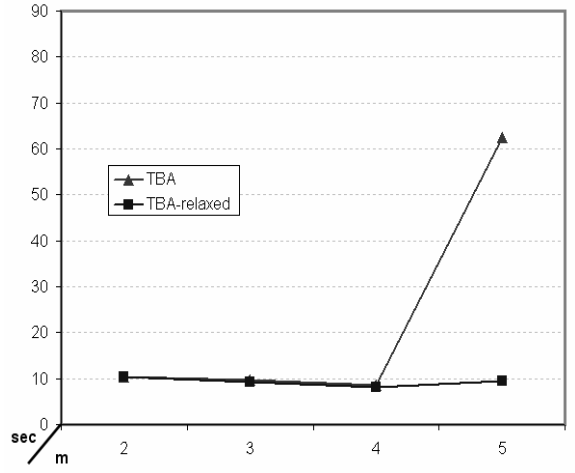

Figure 49: Total time, uniform testbed (100 MB)

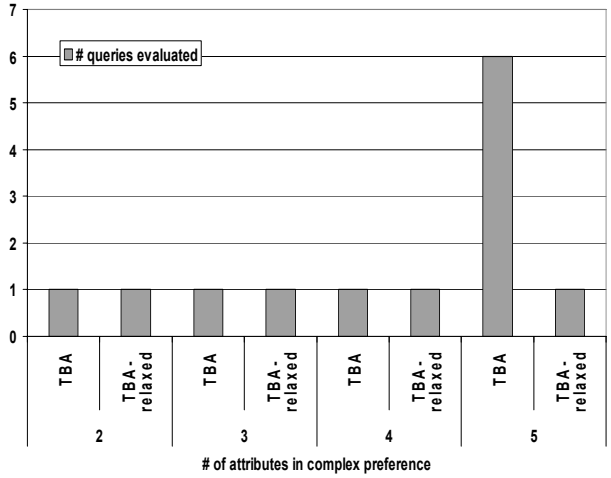

Figure 50: :# queries evaluated - pareto composition

4.9 Effect of the Number of Objects Requested k

In Figures 51 to 54 we report results for our default setting, as a function of k. k was increased such that each increment would result a new block in the answer. Table 7 illustrates the metrics for this experiment in which an 100 MB uniform testbed was used.

Metrics \ k	k=1	k=1.000	k=1.500
$\frac{\lvert V(P,A)\rvert}{\lvert dom(A)\rvert}$	1.728 / 8.000 = 0.216	1.728 / 8.000 = 0.216	1.728 / 8.000 = 0.216
$\frac{\lvert Act(P,A)\rvert}{\lvert R\rvert}$	215.001 / 1.000.000 = 0.215	215.001 / 1.000.000 = 0.215	215.001 / 1.000.000 = 0.215
$\frac{\lvert q_{k,P}(R)\rvert}{\lvert Act(P,A)\rvert}$	754 / 215.001 = 0.004	1.253 / 215.001 = 0.006	1.507 / 215.001 = 0.007
$\frac{\lvert V(P,A)\rvert}{\lvert Act(P,A)\rvert}$	1.728 / 215.001 = 0.008	1.728 / 215.001 = 0.008	1.728 / 215.001 = 0.008

Table 7**:** Metric values (increasing k)

We see that the overall execution time for the algorithms was increased due to the need to process more objects. However, both LBA, TBA still maintain a signifficant advantage over the rest algorithms. Specially, LBA outperforms BNL by 2 orders of magnitude while TBA by 1. BNL is more sensitive in k since in order to construct the next block of the answer, needs to perform another scan over the database and process again all objects (both active and inactive ones).

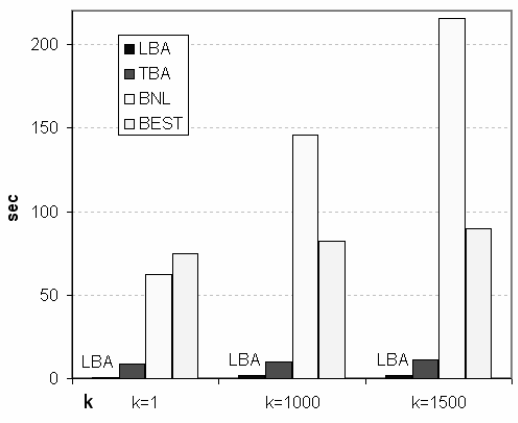

Figure 51: Total time, 100MB uniform testbed

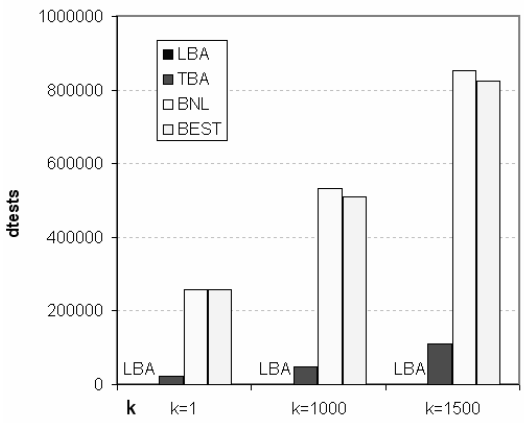

Figure 52: #Dominance tests, 100MB uniform testbed

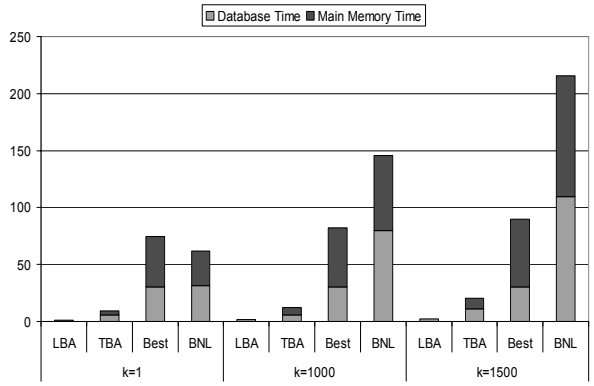

Figure 53: Total time, 100MB uniform testbed

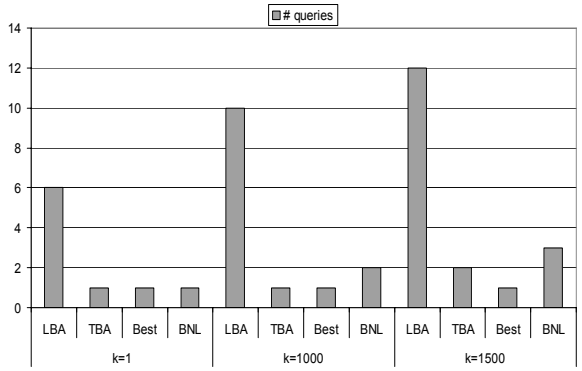

Figure 54: #queries evaluated, 100MB uniform testbed

4.10 Conclusions

	LBA	TBA	$LBA-SQ-conj$
$\frac{\lvert V(P,A)\rvert}{\lvert Act(P,A)\rvert} < 1$	$+$	$-$	$-$
$\frac{\lvert V(P,A)\rvert}{\lvert Act(P,A)\rvert} > 1$ and &	$-$	$-$	$+$
$\frac{\lvert V(P,A)\rvert}{\lvert Act(P,A)\rvert} > 1$ and $\triangleright$	$-$	$+$	$-$

Table 8: Proposed algorithms in various cases

Altogether, we can draw the following conclusions:

- The larger the database gets LBA and TBA become more and more attractive. BNL and Best do not scale well over the database size and that is because at least one scan of the database is required.
- The performance of LBA degrades if $\frac{\lvert V(P,A)\rvert}{\lvert Act(P,A)\rvert} >> 1$ while TBA outperforms LBA in such a scenario especially when the preference expression is prioritized.

- The performance of BNL, Best drops significantly when a large portion of the database objects are active (i.e., the larger the ratio $\frac{|Act(P,A)|}{|R|}$ gets) and that is because at least one dominance test needs to be performed for each active object. LBA is the best algorithm in such cases since its the only algorithm that does not perform dominance tests.
- The $<_{\not\exists\forall}$ variations of LBA, TBA are more efficient than the $<_{\exists\forall}$ variations. Especially $LBA\text{-}SQ\text{-}conj$ is the most efficient $<_{\not\exists\forall}$ variation.
- BNL is more sensitive in k than the rest of the algorithms since in order to construct the next block of the answer, BNL needs to perform another scan over the database.

In summary, we propose that a system should implement algorithms LBA, $LBA\text{-}SQ\text{-}conj$ and TBA. In particular a system should use LBA in cases where $\frac{|V(P,A)|}{|Act(P,A)|}<1$, TBA in cases where $\frac{|V(P,A)|}{|Act(P,A)|}>1$ and the preference expression contains $\triangleright$ operators and $LBA\text{-}SQ\text{-}conj$ in cases of pareto compositions of high dimensionality.

Chapter 5: Related Work

5.1 Related Frameworks for Preference Modelling

In this chapter, we overview the relative approaches for the representation of preferences. Because qualitative approaches are more expressive compared to the quantitative ones and due to the fact that we our framework constitutes also a qualitative approach in this section the most important and expressively richer inquiring works of this category are illustrated, pointing out their main characteristics.

5.1.1 Kiessling's Framework

Kiessling ([13], [15]) defines preferences as strict partial orders over attribute domains. In particular, given $A = \{A_1, \ldots, A_k\}$ a set of attributes A_j with domains $dom(A_j)$, a preference $P = (A, <_P)$ is a strict partial order of $dom(A) = dom(A_1) \times \ldots \times dom(A_k)$, shown as $<_P \subseteq dom(A) \times dom(A)$. For $x, y \in dom(A)$, "$x <_P y$" is interpreted as "I like y better than x". Kiessling for ease of use defines a number of *base preference constructors*. Their goal is to provide intuitive and convenient ways to inductively construct a preference $P = (A, <_P)$. Formally, a base preference constructor has two arguments, the first characterizing the attribute names A and the second the strict partial order $<_P$. The most common constructors include following:

- For *categorical* attributes: POS, NEG, POS/POS, POS/NEG, EXP
- For *numerical* attributes: $AROUND$, $BETWEEN$, $LOWEST$, $HIGHEST$, $SCORE$

POS specifies that a given set of values *should be* preferred. Conversely, NEG states a set of disliked values *should be* avoided if possible. POS/POS and POS/NEG express certain combinations. For example, assuming a preference $P = POS/NEG(\mathrm{A}, POS-set\{v_1, v_2, \ldots, v_m\}, NEG-set\{v_{m+1}, v_{m+2}, \ldots, v_{m+n}\})$ we have $x <_P y$ iff $(x \in NEG-set \wedge y \notin NEG-set) \vee (x \notin NEG-set \wedge x \notin POS-set \wedge y \in POS-set)$ (i.e., a desired value *should be* one from a set of favorites. Otherwise it *should not be* any from a set of dislikes. If this is not feasible too, better than getting nothing any disliked value is acceptable). EXP explicitly enumerates 'better-than' relationships for example $P = EXP(color\{(green, red), (black, yellow)\})$. $AROUND$ prefers values closest to a stated value, $BETWEEN$ prefers values closest to a stated interval. $LOWEST$ and $HIGHEST$ prefer lower and higher values, respectively. $SCORE$ maps attribute values to numerical scores, preferring higher scores.

Kiessling produces more complex preferences by using the following *complex preference constructors*:

- *Pareto* preferences: $P = P_1 \otimes P_2 \otimes \ldots \otimes P_n$. P is a combination of equally important preferences, implementing the pareto-optimality principle.
- *Prioritized* preferences: $P = P_1 \,\&\, P_2 \,\&\, \ldots \,\&\, P_n$. P evaluates *more important* preferences earlier, similar to a lexicographical ordering. P_1 is most important, P_2 next, etc.
- *Numerical* preferences: $P = rank_F(P_1, P_2, \ldots, P_n)$. P combines $SCORE$ preferences P_i by means of a numerical ranking function F.

Kiessling in [19] and [17] was based on the framework described above in order to construct extensions to XPATH and SQL which he calls Preference XPATH and Preference SQL respectively.

Compared to our framework, Kiessling does not seperate between active and inactive objects. Since inactive objects are incomparable to the active ones, he puts them in the set of the undominated (top-1) objects. Moreover, by defining preferences as strict partial orders, the user is not able to define equivalence

relations. For the computation of the top-k objects of a relation, Kiessling introduces a relational operator that he calls *BMO* ([13], [15]). BMO selects the set of the most preferred objects (i.e., the first block), according to a given preference expression. For the evaluation of the BMO operator Kiessling applies *Block Nested Loop* (*BNL*) [13].

5.1.2 Chomicki's Framework

Chomicki, in his work ([7], [8]) emphasizes the view of preferences as first order logical formulas which he calls *preference formulas*. Specifically a preference formula $C(t_i, t_j)$ on $R(A_1,...,A_n)$, where $t_i, t_j \in dom(A_1)\times\cdots\times dom(A_n)$, is a first order logic formula that defines a preference relation $\succ_C \subseteq (dom(A_1)\times\cdots\times dom(A_n))\times(dom(A_1)\times\cdots\times dom(A_n))$ on R as follows:

$$t_i \succ_C t_j \text{ iff } C(t_i, t_j)$$

If $t_i \succ_C t_j$ it means that a tuple t_i dominates a tuple t_j in $\succ_C$. At this point, two important observations need to be made. Firstly, Chomicki does not assume any properties for the preference relations contrary to our framework in which we define preferences as preorders and Kiessling's framework where preferences are considered as strict partial orders. Moreover according to Chomicki, a preference relation is defined directly over the objects of the database contrary to the remaining frameworks where preference relations are defined over attribute domains. Each preference relation $\succ_C$ defines an indifference relation denoted by $\cong_C$ as follows:

$$\forall t_i, t_j \in dom(A_1)\times\cdots\times dom(A_n),\ t_i \cong_C t_j \text{ iff } t_i \not\succ_C t_j \text{ and } t_j \not\succ_C t_i$$

It easy for someone to see that an indifference relation $\cong_C$ actually encapsulates two notions that were defined seperately to our framework. The equivalence relation $\sim$ and the incomparability relation $\|$. Therefore a user can not explicitly define that two or more values are equivalent or incomparable to each other.

Chomicki considers two different kinds of composition for producing more complex preferences. The *undimensional* composition which involves preference relations over just one table and the *multidimensional* one, which involves preference relations defined to more than one tables. The undimensional composition is divided into *boolean* and *prioritized* composition. The most commonly used boolean compositions include *union*, *intersection*, and *set difference* which are defined as follows:

Assume a relation $R(A_1,...,A_n)$ and the preference relations $\succ_C, \succ_{C'}$ on R. Moreover let C_B be a preference formula on R that defines a preference relation $\succ_{C_B}$ on R.

- $\succ_{C_B}$ is the union of $\succ_C, \succ_{C'}$ (denoted by $\succ_C \cup \succ_{C'}$) iff: $t_i \succ_{C_B} t_j \equiv t_i \succ_C t_j \vee t_i \succ_{C'} t_j$
- $\succ_{C_B}$ is the intersection of $\succ_C, \succ_{C'}$ (denoted by $\succ_C \cap \succ_{C'}$) iff: $t_i \succ_{C_I} t_j \equiv t_i \succ_C t_j \wedge t_i \succ_{C'} t_j$
- $\succ_{C_B}$ is the set difference of $\succ_C, \succ_{C'}$ (denoted by $\succ_C - \succ_{C'}$) iff: $t_i \succ_{C_D} t_j \equiv t_i \succ_C t_j \wedge t_i \not\succ_{C'} t_j$

where $t_i, t_j \in dom(A_1) \times \cdots \times dom(A_n)$. Now, a prioritized composition is defined as follows: Assume a relation $R(A_1,...,A_n)$ and the preference relations $\succ_C, \succ_{C'}$ on R. Let $C_\triangleright$ be a preference formula on R that defines a preference relation $\succ_{C_\triangleright}$ on R. $\succ_{C_\triangleright}$ is the prioritized composition of $\succ_C, \succ_{C'}$ denoted by $\succ_C \triangleright \succ_{C'}$ iff: $t_i \succ_{C_\triangleright} t_j \equiv t_i \succ_C t_j \vee (t_i \cong_C t_j \wedge t_i \succ_{C'} t_j)$ where $t_i, t_j \in dom(A_1) \times \cdots \times dom(A_n)$.

On the other hand, the multidimensional composition is divided into the *pareto* and into the *lexicographical* composition which are defined as follows: Assuming relations $R(A_1,...,A_n)$, $S(B_1,...,B_m)$ let $\succ_C$ be a preference relation on R and $\succ_{C'}$ a preference relation on S.

- A preference relation $\succ_{C_P}$ on $R \times S$, is the *pareto composition* of $\succ_C$, $\succ_{C'}$ iff $\forall t_i, t_j \in dom(A_1) \times \cdots \times dom(A_n)$ and $\forall t'_k, t'_l \in dom(B_1) \times \cdots \times dom(B_m)$ it holds $(t_i, t'_k) \succ_{C_P} (t_j, t'_l) \equiv t_i \succeq_C t_j \wedge t'_k \succeq_{C'} t'_l \wedge (t_i \succ_C t_j \vee t'_k \succ_{C'} t'_l)$, where $\forall F \in \{C, C'\}$, $x \succeq_F y \equiv x \succ_F y \vee x \cong_F y$.
- A preference relation $\succ_{C_L}$ on $R \times S$, is the *lexicographical composition* of $\succ_C$, $\succ_{C'}$ iff $\forall t_i, t_j \in dom(A_1) \times \cdots \times dom(A_n)$ and $\forall t'_k, t'_l \in dom(B_1) \times \cdots \times dom(B_m)$ it holds $(t_i, t'_k) \succ_{C_L} (t_j, t'_l) \equiv t_i \succ_C t_j \vee (t_i \cong_C t_j \wedge t'_k \succ_{C'} t'_l)$.

Similar to Kiessling, Chomicki does not seperate between active and inactive objects. Inactive objects will be returned as some of the best objects w.r.t a preference relation. Independently to Kiessling, Chomicki introduces a similar to BMO relational operator that he calls *winnow* ([7], [8]). The framework proposed by Chomicki is very expressive in its principals, yet only a portion of it has been addressed from the implementation perspective; even more all such proposals suggest variations of the dominance testing idea leading to quadratic costs. In [9], a so call weak order framework is introduced, under which a similar to Best, single pass algorithm, for the evaluation of preference queries is proposed. Yet, it requires a very narrow semantics in which all non equal objects of each block are incomparable to each other, and each of them dominates every object of the succeeding block, and is dominated by every object in its preceding block. This requirement is much stricter than those of our framework, employing a much harder to satisfy relation than the $<_{\overline{\forall\exists}}$ relation we introduced.

In [22], [23] a model for representing and storing preferences is proposed. Numerical values between -1 and 1 are used to express the interest, i.e., the preference of a user. However, this seems not an intuitive understandable model. In our framework preferences are modeled in a more natural fashioned way. Furthermore, the algorithms presented in this work bear similarities with

the query rewritings presented in [22], [23] where the integration of personalization into database queries with the use of structured user profiles has been proposed.

It is worth mentioning that there are several different alternatives that define how preference relations order the value space. For instance, [4] distinguishes between *ceteris paribus* and *totalitarian semantics*. According to the ceteris paribus semantics, for a given preference P a tuple t is more preferred than a tuple t' iff $t' <_P t$ and t is equal to t' to every other attribute that describes the tuples (except the ones involved in the preference). As in our work we do not impose the latter constraint, our semantics is totalitarian. To our opinion, ceteris paribus semantics is quite restrictive, and it is unclear if preference queries that follow the ceteris paribus assumption can be evaluted efficiently in large databases and real scale web applications (no paper presents experimental results). CP-nets [3], [10] (that is, Conditional Preference nets) are a graphical model for representation and reasoning about certain sets of qualitative preference statements, interpreted under the ceteris paribus assumption.

5.2 Top-k Algorithms

The top-k paradigm has been first introduced ([11], [12]) in order to reduce the communication cost needed in distributed systems and middleware, in order to aggregate the ranked results coming from several systems. The authors assume that each database consists of a finite set of objects. Each object has m values $x_1, x_2, \ldots, x_m$, where each x_i is actually the score of object o under one of the m attributes. For each x_i it holds $x_i \in [0,1]$ while $f(x_1, x_2, \ldots, x_m)$ is the overall score of object o for an aggregation function f. The database consists of m sorted lists $L_1, L_2, \ldots, L_m$, one for each attribute. Each entry of L_i has the form (o, x_i), where x_i is the i^{th} value of o. Each list L_i is sorted in decreasing order by the x_i value. Also, they consider two modes of access to data: the *sorted* access and the *random* one. A sorted access is a sequential access from the top of a list. Here, the system obtains the score of an object in one of the sorted lists

by proceeding through the list sequentially from the top. Thus, if an object o has the l^{th} highest score in the i^{th} list, then l sorted access are required in L_i to get this score under sorted accesses. In random access, the system requests the score of object o in the L_i list, and obtains it in one random access. Of course, a random access is considered more expensive than a sorted one.

Instead of executing the naive algorithm to obtain the top-k objects (look at every entry in each of the m sorted lists, and compute the overall score of every object), several algorithms have been proposed. At first, Fagin [12] introduced an algorithm, named FA (Fagin's Algorithm). Initially, the FA executes sorted access to each of the m sorted lists L_i in parallel, (i.e., access the top member of each of the lists, then the second member and so on). FA waits until there is a set of at least k objects, such that each of these objects has been seen in each of the m lists. Then for each object that was seen, FA finds the missing values x_i, with a random access to each list L_i. Finally, FA computes the overall scores according to the aggregation function f for all objects that have been seen and returns the objects with the k highest scores. The Threshold Algorithm (TA) [11] is an enhancement over FA. TA performs sorted accesses in parallel to each of the m sorted lists L_i. For each object o seen, TA performs random accesses to the other lists to find the score x_i of o in every list L_i and then computes the overall score of o. For each list L_i let $\overline{x}_i$ be the score value of the last object seen under sorted access. TA computes a threshold value t to be $f(\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_m)$ and works as an upper bound for the unseen objects. The algorithm stops when at least k objects have been seen whose score is at least equal to t and returns the k objects with the highest scores. Quite similar to TA are also algorithms Multi-step [26] and Quick-Combine [20].

Furthermore, No Random Access Algorithm (NRA) [11] is proposed for systems where random access to the ranked lists is not allowed. NRA performs only sorted accesses in parallel to each of the m sorted lists L_i. At each depth d (i.e., when the first d objects have been visited across all m lists) the bottom

values $\underline{x}_1^{(d)}, \underline{x}_2^{(d)}, \ldots, \underline{x}_m^{(d)}$ are maintained as the scores last seen from each input list. For every object o NRA computes a lower bound $W^{(d)}(o)$ and an upper bound $B^{(d)}(o)$. The lower bound for an object o at depth d is the score of the aggregate function f where for each unknown score x_i we put 0. In the computation of the upper bound for each unknown score x_i we put the value $\underline{x}_i^{(d)}$. The algorithm maintains the k objects with the largest $W^{(d)}$ (ties are broken using an object's $B^{(d)}$ score). Let $M_k^{(d)}$ be the k^{th} largest worst score. Then NRA stops when k distinct objects have been seen and all the other objects outside the top-k objects have an upper bound value less or equal to $M_k^{(d)}$. Quite similar to NRA are also Stream-Combine [18] and SR-Combine [16].

Finally in [25], the authors introduced Algorithms Upper and Pick for evaluating top-k selection queries over web-accessible sources assuming that only random access is available for a subset of the sources. Similarly, Algorithm MPro by Chang and Hwang [6] addresses the expensive probing of some of the object scores in top-k selection queries. They assume a sorted access on one of the attributes while other scores are obtained through probing or by executing a user-defined function on the remaining attributes. Unlike to the algorithms presented above, which take the data locality parameter into account, our work assumes that all data are locally available, thus fetching a tuple implies that all attribute values are fetched at once as well.

5.3 Skyline Algorithms

Assuming a set D of n-dimensional data objects $o = (o_1, \ldots, o_n)$ and n score functions $S = (s_1, \ldots, s_n)$, a *domination* relation (denoted by $<_S$) is defined over the elements of D as follows:

- $o' <_S o$: iff $\exists i \in [1 \ldots n]$, such that $s(o_i') < s(o_i)$ and $\forall j \in [1 \ldots n] - \{i\}$ it holds $s(o_j') \leq s(o_j)$

The *skyline* [29] is defined as those objects of a relation that are not dominated by any other object. An object dominates another object if it is as good or better in all dimensions and better in at least one dimension. (i.e., $Skyline_S(D) = \{o \in D \mid \nexists o' \in D \; s.t. \; o <_S o'\}$). Skyline queries are directly and naturally related to the case where all atomic preferences have been composed using pareto preferences, each atomic preference is a total order of values and the number of requested objects equals to 1 (top-1 objects). Several algorithms for computing the skyline have been proposed. These can be categorized into *non-index based* (e.g. BNL [29]) and *index based* (e.g. NN [23], BBS [25]). Since BNL was already fully described in Chapter 3, below we describe the index based skyline algorithms.

Kossmann et al. presented in [21] a progressive skyline algorithm (NN). Based on Nearest Neighbor queries, their algorithm continuously returns skyline points. Unfortunately, this algorithm has problems with high dimensional spaces (because of multiple access to the same node, duplicate elimination has to be performed). Furthermore, as shown in [27], this algorithm has a huge space overhead; a used data structure could reach the size of the whole data set. An improved algorithm called BBS (branch-and-bound skyline) for processing progressive skyline queries in a local scenario was presented by Papadias et al. in [27]. Like NN, that algorithm is based on Nearest Neighbor queries. It uses a multidimensional indexing method, such like an R-tree. The minimal distance to the point of origin (w.r.t. a score function that is monotonic on each attribute) is assigned to all minimum bounding boxes in the R-tree. At the beginning of the algorithm, the root entries of the tree are added to a heap structure that sorts its entries based on their minimum distances. In parallel, a list containing all possible skyline points S is maintained (initially, the list is set to the empty set). The algorithm successively removes all bounding boxes b from the heap. If b is dominated by any point that is already in S, b is discarded immediately. Otherwise, b's children are processed one after another: If the child is again a compound entry, it is added to the heap structure unless is dominated by any

skyline point found so far. If the child is a point, it is added to S. Once the heap is empty, S contains the correct skyline.

As expected the non index-based algorithms are typically inferior to the index-based ones. However, all these algorithms require appropriate indexes built on the skyline dimensions. In particular, they require to build (beforehand or on the fly) indexes over any of the non-empty subsets of a d-dimensional set of a relation R with d attributes to accommodate $2^d - 1$ different skyline queries. On the contrary, our work assumes only d indexes (i.e., one index for each attribute). Moreover, all these index-based algorithms handle only totally ordered attribute domains and it is unclear if they can still maintain their competitiveness for partially (pre)ordered preferences.

Recently, the problem of evaluating skyline queries with partially-ordered domains was studied in [5]. The proposed solution relies on graph encoding techniques to transform a partial ordered domain into two total orders (using interval-based labels) and thus exploit index-based algorithms for computing skyline queries on the transformed space. We believe that the linearization of partial preorders we propose in this paper based on cover relations provides a natural semantics for evaluating arbitrary preference queries (and not only the fragment of skyline queries) whereas avoids the computation costs of generating and maintaining interval-based labels for graphs. Furthermore, even for small sized databases (500 and 1000k tuples of unspecified size), the experimental evaluation presented in [5] demonstrates that the proposed algorithms do not scale well, when the majority of the attributes that are involved are partially-ordered. For example, for 2 totals and 1 partially ordered attributes a typical time of almost 50 sec, whereas, for 1 total and 2 partial order attributes this time rises above 1200 seconds. No results are presented for more than 2 partially ordered attributes. In our case, the algorithms we introduced scale much better w.r.t. the number and nature of the involved preferences.

Chapter 6: Conclusion and Future Work

Enabling users to quickly focus on the k best results according to their specified needs and preferences is essential for several modern applications. In this thesis, we elaborated the problem of computing the top-k objects for the case where user preferences are expressed qualitatively (i.e., as non-antisymmetric preorders). Initially, we presented existing algorithms and demostrated their deficiencies, which severely limit their applicability. Subsequently, we introduced two novel progressive algorithms called LBA and TBA that follow a query-based ordering approach for the evaluation of the top-k objects. The intuition of the query-based ordering is to use the specified user preferences for defining an ordering over queries that need to be evaluated in order to retrieve the top-k objects.

In particular, LBA defines an ordering over queries which are essentially conjunctions of atomic selection conditions, over all attributes that the user preferences involve. The algorithm does not perform dominance tests over objects and accesses only the top-k objects and only once. In a similar fashion, but from a different angle, TBA defines an order of queries which are disjunctions of atomic selection conditions over a single attribute that the user preference involves. TBA uses appropriate threshold values and takes into account the selectivities of the atomic selection conditions in order to avoid fetching more objects than those actually required. However TBA will access not only the top-k objects but also a portion of the active and inactive ones and probably more than once while dominance tests are performed, but only for the small number of the retrieved objects.

We compared the algorithms analytically and we described the cases where each of them is expected to outperform the rest. Furthermore, we defined a relaxation of the classical definition of top-k objects for a rise in efficiency and presented some variations of our proposed algorithms. Finally, we

systematically used various experimental evaluation settings to demonstrate the effectiveness of the algorithms we introduced and illustrate their superior performance.

The top-k algorithms we have introduced take as input a preference expression P and an integer k and return the top-k objects of an object relation R. An interesting path of exploration involves modifing these algorithms in order to evaluate efficiently *preference based queries*, i.e., queries that contain both a regular (filtering) query part and a preference part. Formally, a preference based query over an object relation R is a triple $Q_{PB} := (q_r, P, k)$ where:

- q_r is a regular query, providing filtering conditions
- P is a preference relation
- k is a positive integer indicating a top-k answer request

Let $Ans(q_r)$ denote the answer of the regular query q_r of a preference based query Q_{PB}; $Ans(q_r)$ consists of a set of unordered objects of R. Consequently, the corresponding preference query $Q := (P, k)$ of Q_{PB} should return an ordered subset of $Ans(q_r)$, which will comprise the answer $Ans(Q_{PB})$ to the preference based query.

We can modify our top-k algorithms, to evaluate preference based queries in several ways. One approach is to append the filtering conditions of q_r into each of the Query Lattice queries that our algorithms construct and evaluate. For example, assume that at some point TBA needs to evaluate query $A_1 = a_1 \vee A_1 = a_2$ and suppose that the preference based query adds a filtering part $q_r := A_2 = b \wedge A_3 = c$; these filtering conditions may be integrated into the former query, and, thus, TBA will construct and evaluate the updated query $(A_1 = a_1 \vee A_1 = a_2) \wedge A_2 = b \wedge A_3 = c$. It is obvious that the execution plan of this updated query will change; the impact of this change on the overall algorithm performance is not a priori known, and, besides the possible additional attributes and indexes, it is the DBMS, with its optimization techniques, that constitutes a critical factor for it. An alternative approach exploits the idea of applying the algorithms on $Ans(q_r)$, rather than on R. This implies that $Ans(q_r)$ is

evaluated and materialized first, and, as a second step, our algorithms are applied on $Ans(q_r)$.

In our framework, we rely on unconditional, positive preferences for the presence of values over attributes of a single relational table. As part of our future work, we plan to enhance our framework with some interesting extensions such as combining preferences through joins for evaluating preferences over several tables, allowing preferences to appear more than once in a preference expression and supporting conditional preferences. Preferences on the absence of values, as well as negative ones, 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). One final remark concerns inactive objects. We assumed that there are at least k active objects in a database with respect to some preference expression P. However, when the set of active objects turns out to be relatively small (w.r.t. k), then one may wish to include some inactive objects in the answer as well. In this respect, objects that are active with respect to a bigger subset of atomic preferences, or with respect to atomic preferences over more important attributes, as defined by the user, could be considered as candidate objects to include in the result, in order to reach the number k which was requested.

Bibliography

[1] R. Agrawal and E. L. Wimmers.: A Framework for Expressing and Combining Preferences. In *Proceedings of the ACM SIGMOD*, pages 297-306, Dallas, 2000.

[2] Wolf-Tilo Balke and Ulrich Guntzer: Multi-Objective Query Processing for Database Systems. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 936–947, Toronto, Canada, September 2004.

[3] Boutilier, C., Brafman, R., Hoos, H., and Poole, D. Reasoning with conditional ceteris paribus preference statements. In *UAI-99*, pages 71-80, 1999.

[4] R. Brafman and C. Domshlak: *Database Preference Queries Revisited*. Technical Report TR2004-1934, Cornell University Computing and Information Science, 2004.

[5] Chee-Yong Chan, Pin-Kwang Eng, and Kian-Lee Tan: Stratified Computation of Skylines with Partially-Ordered Domains. In *Proceedings of the ACM SIGMOD*, pages 203 - 214, Baltimore, 2005.

[6] Kevin Chen-Chuan Chang and Seung won Hwang: Minimal Probing: Supporting expensive predicates for top-k queries. In *Proceedings of the ACM SIGMOD*, Madison, Wisconsin, 2002.

[7] Jan Chomicki: Querying with intrinsic preferences. In *Proceedings of the 8th International Conference on Extending Database Technology (EDBT)*, pages 34-51, London, UK, 2002. Springer-Verlag.

[8] Jan Chomicki: Preference formulas in relational queries. *ACM Trans. Database Syst.*, 28(4):427-466, 2003.

[9] J. Chomicki: Semantic Optimization of Preference Queries. In *1st International Symposium on Constraint Databases*, pages 133-148, 2004.

[10] Domshlak C. and Brafman R. Cp-nets - reasoning and consistency testing. In *KR-02*, pages 121-132, 2002.

[11] R. Fagin, A. Lotem, and M. Naor: Optimal Aggregation Algorithms for Middleware. In *Proceedings of the 2001 ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 2001.

[12] Ronald Fagin: *Combining Fuzzy Information From Multiple Systems*. Journal of Computer and System Sciences, 58(1):83-99, 1999.

[13] B. Hafenrichter and W. Kiessling: Optimization of Relational Preference Queries. In *Proceedings of the Sixteenth Australasian Database Conference (ADC)*, pages 175-184, Newcastle, Australia, 2005.

[14] V. Hristidis, N. Koudas, and Y. Papakonstantinou: PREFER: A System for the Efficient Execution of Multiparametric Ranked Queries. In *Proceedings of the the ACM SIGMOD*, pages 259-269, Santa Barbara, USA, 2001.

[15] W. Kiessling: Foundations of Preferences in Database Systems. In *Proceedings of the 28th Intern. Conf. on Very Large Data Bases (VLDB)*, pages 311-322, Hong Kong, China, 2002.

[16] Kiessling W. Balke W.-T., Guntzer U: On Real-time Top k Querying for Mobile Services. In *International Conference on Cooperative Information Systems*, Irvine, USA, 2002.

[17] W. Kiessling and G. Kostler: Preference SQL - Design, Implementation, Experiences. In *Proceedings of the 28th Intern. Conf. on Very Large Data Bases (VLDB)*, pages 990-1001, Hong Kong, China, 2002.

[18] Kiessling W. Guntzer U., Balke W.-T: Towards efficient multi-feature queries in heterogeneous enviroments. In *International Conference on Information Technology*, Las Vegas, USA, 2001.

[19] Kiessling W., Hafenrichter B., Fischer S., Holland S.: Preference XPATH: A Query Language for E-Commerce. In *Proceedings of the 5th Intern. Conference on Wirtschaftsinformatik*, Ausburg, Germany, 2001, pp. 427 - 440.

[20] Kiessling W. Guntzer U., Balke W.-T: Optimizing Multi-Feature Queries for Image Databases. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, Cairo, Egypt, 2000.

[21] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, Hong Kong SAR, China, 20–23 August 2002, pages 275–286, Los Altos, CA 94022, USA, 2002.

[22] Koutrika G, Ioannidis Y, Personalized Queries under a Generalized Preference Model. *ICDE* 2005: 841-852

[23] G. Koutrika and Y. Ioannidis.: Personalization of Queries in Database Systems. In *Proceedings of the 20th International Conference on Data Engineering*, Boston, USA, pages 597-608, 2004.

[24] M. Lacroix and P. Lavency.: Putting More Knowledge Into Queries. In *Proceedings of the 13rd International Conference on Very Large Data Bases (VLDB)*, pages 217-225, Brighton, England, 1987.

[25] Marian A. Bruno N., Gravano L: Evaluating Top-k Queries over Web-Accessible Databases. In *International Conference on Data Engineering (ICDE)*, Heidelberg, 2002.

[26] Surya Nepal and M. V. Ramakrishna.: Query processing issues in image (multimedia) databases. In *Proceedings of the 15th International Conference on Data Engineering (ICDE)*, Sydney, Australia, pages 22-29, 1999.

[27] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An optimal and progressive algorithm for skyline queries. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 467-478, New York, NY, USA, 2003.

[28] N. Spyratos and V. Christophides. Querying with preferences in a digital library. In *Dagstuhl Seminar No 05182, Federation over the Web*, May 2005.

[29] Stocker K. Brzsnyi S., Kossman D.: The Skyline Operator. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, Heidelberg, 2001.

[30] Riccardo Torlone and Paolo Ciaccia: Which Are My Preferred Items?. In *Workshop on Recommendation and Personalization in eCommerce*, pages 1-9, Malaga, Spain, 2002.

[31] Yidong Yuan, Xuemin Lin, Qing Liu, Wei Wang, Jeffrey Xu Yu, and Qing Zhang: Efficient computation of the skyline cube. In *Proceedings of the 31st International Conference on Very large Data Bases (VLDB)*, pages 241-252, 2005.