Εξατομικευμένη Διαχείριση Δεδομένων: Μοντέλα, Αλγόριθμοι και Συστήματα

Η ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Πληροφορικής Εξεταστική Επιτροπή

από τον

Κωνσταντίνο Στεφανίδη

ως μέρος των Υποχρεώσεων για τη λήψη του

ΔΙΔΑΚΤΟΡΙΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

Δεκέμβριος 2009

Τριμελής Συμβουλευτική Επιτροπή

- Βασίλειος Δημακόπουλος, Επίκουρος Καθηγητής του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων.
- Μανόλης Κουμπαράκης, Αναπληρωτής Καθηγητής του Τμήματος Πληροφορικής και Τηλεπικοινωνιών του Εθνικού και Καποδιστριακού Πανεπιστημίου Αθηνών.
- Ευαγγελία Πιτουρά, Αναπληρώτρια Καθηγήτρια του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων (επιβλέπουσα).

Επταμελής Εξεταστική Επιτροπή

- Παναγιώτης Βασιλειάδης, Επίκουρος Καθηγητής του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων.
- Βασίλειος Δημακόπουλος, Επίκουρος Καθηγητής του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων.
- Μανόλης Κουμπαράκης, Αναπληρωτής Καθηγητής του Τμήματος Πληροφορικής και Τηλεπικοινωνιών του Εθνικού και Καποδιστριακού Πανεπιστημίου Αθηνών.
- Ιωάννης Μανωλόπουλος, Καθηγητής του Τμήματος Πληροφορικής του Αριστοτελείου Πανεπιστημίου Θεσσαλονίκης.
- Σταύρος Νικολόπουλος, Καθηγητής του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων.
- Ευαγγελία Πιτουρά, Αναπληρώτρια Καθηγήτρια του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων (επιβλέπουσα).
- Βασίλης Χριστοφίδης, Καθηγητής του Τμήματος Επιστήμης Υπολογιστών του Πανεπιστημίου Κρήτης.

Acknowledgements

I would like to express my sincere thanks and gratitude to my advisor Prof. Evaggelia Pitoura for the valuable guidance, advice and encouragement she has offered while supervising my dissertation. Collaborating with her has been a pleasant and memorable experience. Many thanks also to both my advisor and Prof. Ioannis Fudos for their actions for the funding of my research through the program PENED 2003 of the Greek General Secretariat of Research and Technology.

Furthermore, I am grateful to Prof. Panos Vassiliadis for his useful comments and suggestions during my studies at the University of Ioannina. I am also greatly indebted to the members of my supervising committee Prof. Vassilios Dimakopoulos and Prof. Manolis Koubarakis and also, Prof. Vassilis Christophides, Prof. Yannis Manolopoulos and Prof. Stavros Nikolopoulos for their helpful comments and insightful remarks.

I would also like to thank Dr. Georgia Koutrika for our great collaboration during the last year. With Georgia Koutrika, we have worked together on classifying the various approaches that deal with preferences in databases. Moreover, I would like to thank all the people of the DMOD Laboratory who over the course of this research provided me with helpful feedback and turned my time there into a joyful experience. Special thanks to my office-mate and friend Marina Drosou. With Marina Drosou, we have worked together on personalizing keyword search and publish/subscribe delivery. Finally, I would like to thank my parents and my friends for their continuous support and understanding throughout all the years of my studies.

This thesis is part of the 03ED591 research project, implemented within the framework of the "Reinforcement Programme of Human Research Manpower" (PENED) and co-financed by National and Community Funds (20% from the Greek Ministry of Development-General Secretariat of Research and Technology and 80% from E.U.-European Social Fund).

Table of Contents

List of Figures

List of Tables

List of Algorithms

Περιληψη

Κωνσταντίνος Στεφανίδης.
PhD, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων. Δεκέμβριος, 2009.
Τίτλος Διατριβής: Εξατομικευμένη Διαχείριση Δεδομένων: Μοντέλα, Αλγόριθμοι και Συστήματα.
Επιβλέπουσα: Ευαγγελία Πιτουρά.

Λόγω του μεγάλου και συνεχώς αυξανόμενου όγκου δεδομένων που είναι σήμερα διαθέσιμος για κάθε χρήστη, δημιουργείται η ανάγκη του εντοπισμού των πιο σημαντικών δεδομένων για τον ίδιο. Τα συστήματα εξατομίκευσης στοχεύουν στην αντιμετώπιση αυτού του προβλήματος. Χρησιμοποιώντας προτιμήσεις επιτρέπουν στους χρήστες να εκφράσουν το ενδιαφέρον τους σε συγκεκριμένα δεδομένα. Αντικείμενο αυτής της διατριβής είναι: (i) η ανάπτυξη ενός ολοκληρωμένου μοντέλου προτιμήσεων και (ii) η ενσωμάτωση προτιμήσεων σε συστήματα διαχείρισης δεδομένων.

Αρχικά, εστιάζουμε στην έκφραση ενός μοντέλου προτιμήσεων συμφραζόμενου περιβάλλοντος. Μία προτίμηση συμφραζόμενου περιβάλλοντος είναι μία προτίμηση εμπλουτισμένη με πληροφορία σχετική με τις συνθήκες ή το συμφραζόμενο περιβάλλον κάτω από το οποίο αυτή ισχύει. Μοντελοποιούμε το συμφραζόμενο περιβάλλον ως ένα σύνολο από γνωρίσματα που μπορούν να παίρνουν τιμές από διαφορετικά επίπεδα ιεραρχίας. Μία προτίμηση συμφραζόμενου περιβάλλοντος ορίζεται ως μία τριάδα (C, P, S), όπου: (i) το C εκφράζει συνθήκες στις τιμές των γνωρισμάτων του συμφραζόμενου περιβάλλοντος, (ii) το P εκφράζει συνθήκες στις τιμές των γνωρισμάτων του σχήματος μίας βάσης δεδομένων και (iii) το S εκφράζει ένα βαθμό ενδιαφέροντος. Η ερμηνεία μίας τέτοιας προτίμησης είναι ότι, για τις συνθήκες που καθορίζονται από το C, ανατίθεται στις πλειάδες του στιγμιοτύπου της βάσης δεδομένων, που ικανοποιούν τις συνθήκες του P, ο βαθμός ενδιαφέροντος S.

Δοθείσης μίας ερώτησης, το πρόβλημα είναι ο εντοπισμός των προτιμήσεων εκείνων που το συμφραζόμενο περιβάλλον τους είναι πιο σχετικό με το συμφραζόμενο περιβάλλον της ερώτησης. Για την επίλυση του προβλήματος αυτού, ορίζουμε κατάλληλες μετρικές απόστασης. Για να επιταχύνουμε τη διαδικασία εντοπισμού των σχετικών προτιμήσεων, προτείνουμε δομές και αλγορίθμους που εκμεταλλεύονται την ιεραρχική φύση των τιμών του συμφραζόμενου περιβάλλοντος. Έχοντας εντοπίσει τις σχετικές με μία ερώτηση προτιμήσεις, τις χρησιμοποιούμε για τη διαβάθμιση των αποτελεσμάτων της.

Για την αποδοτική διαβάθμιση των αποτελεσμάτων μίας ερώτησης, προτείνουμε μία προσέγγιση η οποία βασίζεται στον προϋπολογισμό βαθμών ενδιαφέροντος για τις πλειάδες της

βάσης δεδομένων. Ακολουθώντας μία ακραία περίπτωση, θα μπορούσαμε να προϋπολογίσουμε για κάθε πλειάδα όλους τους βαθμούς ενδιαφέροντος, για κάθε πιθανή κατάσταση συμφραζόμενου περιβάλλοντος. Επειδή το πλήθος των διαφορετικών καταστάσεων είναι πολύ μεγάλο, υπολογίζουμε βαθμούς ενδιαφέροντος μόνο για συγκεκριμένους αντιπροσώπους των καταστάσεων συμφαζόμενου περιβάλλοντος. Για τον καθορισμό των αντιπροσώπων συμφραζόμενου περιβάλλοντος προτείνουμε δύο μεθόδους. Και στις δύο μεθόδους, αρχικά ομαδοποιούμε τις όμοιες προτιμήσεις και έπειτα προσδιορίζουμε τον αντιπρόσωπο της ομάδας. Στην πρώτη περίπτωση, θεωρούμε ως όμοιες τις προτιμήσεις που εκφράζονται για όμοιο συμφραζόμενο περιβάλλον, ενώ στη δεύτερη, αυτές που οδηγούν σε όμοιους βαθμούς ενδιαφέροντος για τις πλειάδες της βάσης δεδομένων, δηλαδή τις προτιμήσεις με όμοια P και S. Αυτή η μέθοδος βασίζεται σε μία νέα απεικόνιση των προτιμήσεων που χρησιμοποιεί ένα πίνακα δυαδικών ψηφίων το μέγεθος του οποίου εξαρτάται από την επιθυμητή ακρίβεια της διαβάθμισης των αποτελεσμάτων. Η πρώτη μέθοδος μπορεί να εφαρμοστεί τόσο σε ποιοτικές όσο και ποσοτικές προτιμήσεις, ενώ η δεύτερη, επειδή χρησιμοποιεί το βαθμό ενδιαφέροντος των προτιμήσεων, μπορεί να εφαρμοστεί μόνο σε ποσοτικές προτιμήσεις.

Στη συνέχεια, εξετάζουμε πώς οι προτιμήσεις μπορούν να χρησιμοποιηθούν σε δύο σημαντικές εφαρμογές της διαχείρισης δεδομένων. Συγκεκριμένα, μελετάμε την εφαρμογή τους σε αναζητήσεις με βάση λέξεις-κλειδιά και σε συστήματα έκδοσης/συνδρομής.

Η αναζήτηση με βάση λέξεις-κλειδιά είναι πολύ δημοφιλής γιατί επιτρέπει στους χρήστες να εκφράσουν τις ερωτήσεις τους χωρίς να γνωρίζουν τη δομή των δεδομένων ή κάποια γλώσσα ερωτήσεων. Στη διατριβή αυτή, προτείνουμε τη χρήση προτιμήσεων για την υποστήριξη αναζήτησης με βάση λέξεις-κλειδιά, έτσι ώστε διαφορετικοί χρήστες να λαμβάνουν διαφορετικά αποτελέσματα με βάση τα προσωπικά τους ενδιαφέροντα. Για τη διαβάθμιση των αποτελεσμάτων μίας ερώτησης λέξης-κλειδιού, συνδυάζουμε την διάταξή τους, όπως προκύπτει από τις προτιμήσεις του χρήστη, με τη συνάφειά τους με την ερώτηση. Η συνάφεια καθορίζεται από την απόσταση των λέξεων-κλειδιών, όπως προκύπτει από κατάλληλες συνενώσεις, στο παρόν στιγμιότυπο της βάσης δεδομένων. Πέρα από τις προτιμήσεις και τη συνάφεια, προτείνουμε τον υπολογισμό των k αποτελεσμάτων που επιδεικνύουν κάποια διαφορετικότητα και καλύπτουν πολλές προτιμήσεις, από τα αποτελέσματα που είναι πιο συναφή και προτιμητέα. Για τον αποδοτικό υπολογισμό των αποτελεσμάτων μίας ερώτησης λέξης-κλειδιού, προτείνουμε τον αλγόριθμο *Διαμοιρασμού-Αποτελέσματος*, ο οποίος, βασιζόμενος στο ότι τα αποτελέσματα μίας ερώτησης σχετίζονται με τα αποτελέσματα ερωτήσεων υπερσυνόλου, αποφεύγει επαναληπτικούς υπολογισμούς.

Για να ελέγξουμε το πλήθος των γεγονότων που λαμβάνει ένας χρήστης σε ένα σύστημα έκδοσης/συνδρομής, προτείνουμε ένα μηχανισμό διαβάθμισης των γεγονότων ανάλογα με το πόσο σημαντικά είναι αυτά για το χρήστη. Για να διαβαθμίσουμε τα γεγονότα βασιζόμαστε στις προτιμήσεις που έχει εκφράσει ο χρήστης ανάμεσα στις συνδρομές του: ένα γεγονός που ταιριάζει με μία συνδρομή υψηλής προτεραιότητας, θεωρείται σημαντικότερο από ένα γεγονός που ταιριάζει με μία συνδρομή χαμηλότερης προτεραιότητας. Έτσι, ένας χρήστης αντί να λαμβάνει όλα τα γεγονότα που ταιριάζουν με τις συνδρομές του, λαμβάνει μόνο τα k πιο ενδιαφέροντα. Επειδή η δημοσίευση γεγονότων είναι συνεχής, εξετάζουμε

διαφορετικές πολιτικές αποστολής γενονότων, ανάλογα με το εύρος των γεγονότων από το οποίο επιλέγονται τα κορυφαία k γεγονότα. Επειδή ένας χρήστης συχνά επιθυμεί να λαμβάνει γεγονότα που επιδεικνύουν κάποια διαφορετικότητα, ακόμη και αν αυτά δεν αποτελούν γεγονότα της υψηλότερης προτίμησής του, προτείνουμε έναν αλγόριθμο υπολογισμού των k γεγονότων ο οποίος λαμβάνει υπόψη, εκτός από το πόσο σημαντικά είναι, και τη διαφορετικότητά τους από τα υπόλοιπα υψηλά διαβαθμισμένα γεγονότα.

Abstract

Kostas Stefanidis.
PhD, Computer Science Department, University of Ioannina, Greece. December, 2009.
Title of Dissertation: Personalized Data Management: Models, Algorithms and Systems.
Supervisor: Evaggelia Pitoura.

Today, there is a large amount of information available for every user. Locating the most valuable or important information can prove an overwhelming task due to the huge volume of accessible data. Personalization systems aim at tackling this problem by utilizing preferences to allow users to express their interest on specific pieces of data. The goal of this thesis is the study of preferences in data management systems. To this end, we first introduce a context-dependent model for preferences and appropriate data structures and algorithms for managing contextual preferences. Then, we explore the integration of preferences to achieve personalized ranked retrieval (*i*) in keyword search in database systems and (*ii*) in publish/subscribe data delivery.

We define contextual preferences as preferences annotated with specifications regarding the context under which they hold. We propose modeling context as a set of multidimensional context parameters that take values from hierarchical domains. A contextual preference is defined as a triple (C, P, S), where: (*i*) C expresses conditions on the values of the context parameters, (*ii*) P expresses conditions on the values of the attributes of the database schema and (*iii*) S expresses a degree of interest. The meaning of such a contextual preference is that, under all circumstances specified by C, all database tuples that satisfy the conditions of P are assigned the indicated degree of interest S.

Given the context that a query is associated with, the problem is to identify those preferences that are applicable to the context that is the most relevant to the context of the query. This problem, called *context resolution problem*, can be distinguished between: (*i*) the identification of all candidate contexts that encompass the query context and (*ii*) the selection of the most appropriate contexts among these candidates. The first sub-problem is resolved through the notion of cover partial order between contexts that relates contexts expressed at different levels of abstraction. To resolve the second sub-problem, we employ distance metrics that capture similarity between contexts.

We introduce algorithms for context resolution that build upon two data structures, namely the *preference graph* and the *profile tree*, that index preferences based on their associated context. The preference graph explores the cover partial order of contexts to

organize them in some form of a lattice. A top-down traversal of the graph supports an incremental specialization of a given context, whereas a bottom-up traversal incremental relaxation. The profile tree offers a space-efficient representation of contexts by taking advantage of the co-occurrence of context values in preferences. It supports exact matches of contexts very efficiently through a single root-to-leaf traversal.

To efficiently assign scores to database tuples using contextual preferences, we introduce a method based on pre-computing tuple scores. At one extreme, we could compute all different scores for each tuple for all potential contexts. However, since the number of contexts grows rapidly with the number of context parameters, we pre-compute scores for representative contexts. We propose two complementary approaches to defining representative contexts. The first approach clusters together similar contexts and selects the resulting cluster descriptions as representative contexts. Context similarity exploits the hierarchical nature of the domain of the context parameters. The other approach groups together preferences that would result in similar scores for all database tuples. This method takes advantage of the quantitative nature of preferences to group together those preferences that have similar predicates and scores. The method is based on a novel representation of preferences through a predicate bitmap table whose size depends on the desired precision for the resulting scoring.

Next, we study how preferences can be explored to achieve personalization in keyword search and in publish/subscribe systems. Keyword search is very popular, because it allows users to express their information needs without either being aware of the underlying structure of the data or using a query language. In this thesis, we propose personalizing keyword search in relational database systems. Since keyword search is often best-effort, given a budget k on the number of results, we combine the order of results as indicated by the user preferences with their relevance to the query. Besides preferences and relevance, we consider the set of the k results as a whole and seek to increase the overall importance of this set to the users. Our goal is to select the k results that both cover different preferences and exhibit small overlap among the relevant and preferred results. We introduce a *Sharing-Results* keyword query processing algorithm, that exploits the fact that the results of a keyword query are related with the results of its superset queries, to avoid redundant computations. We also propose an algorithm that works in conjunction with the Sharing-Results algorithm to compute the top-k representative results.

To control the rate of notifications received by the users in a publish/subscribe system, we propose extending subscriptions to allow users to express the fact that some events are more important to them than others. In particular, we introduce *preferential subscriptions* by formulating preferences among subscriptions. Events are ranked, so that, an event that matches a highly preferred subscription is ranked higher than an event that matches a subscription with a lower preference. We propose a top-k variation of the publish/subscribe paradigm in which users receive only the matching events having the k highest ranks instead of receiving all events matching their subscriptions. Since the top-k events are often very similar to each other, we adjust the top-k computation to take also

into account the diversity of the delivered events. Finally, since the generation of events is continuous, we study a number of delivering policies that determine the range of events over which the top-k computation is performed.

Chapter 1

Introduction

Preferences guide human decision making from early childhood (e.g. "which ice cream flavor do you prefer?") up to complex professional and organizational decisions (e.g. "which investment funds to choose?"). Preferences have traditionally been studied in philosophy, psychology and economics and applied to decision making problems. For instance, in philosophy, they are used to reason about values, desires and duties ([56]). In mathematical decision theory, preferences (or utilities) model economic behavior ([50]). The notion of preferences has in recent years drawn new attention from researchers in other fields, such as artificial intelligence, where they capture agent goals ([21, 38, 124]), and databases, where they capture soft criteria for database queries. Explicit preference modeling provides a declarative way for choosing among alternatives, whether these are solutions of problems to solve, answers of database queries, decisions of a computational agent, plans of a robot and so on.

In databases, interest in preferences was triggered by observing the limitations of the Boolean database answer model, where query criteria are considered as *hard* or *mandatory* by default and a non-empty answer is returned only if it satisfies all the query criteria. In this context, a user can face either of two problems: (*i*) the *empty-answer* problem, when the conditions are too restrictive or the data cannot exactly match the query or (*ii*) the *too-many-answers* problem, where too many results match the query. It is hard for users to cope with these problems especially when accessing databases on the web, where the schema and contents of the database are not known and users are unlikely to be familiar with a structured query language in order to formulate more accurate queries.

Incorporating soft query criteria or preferences in a query can help cope with these problems. The empty-answer problem can be tackled by relaxing some of the hard constraints in the query, i.e. considering them as soft or as user wishes (e.g. [74]) or even

by replacing them by other constraints that capture user preferences related to the given query context and returning results that are ranked according to how well they match the modified query. The too-many-answers problem can be tackled by strengthening the query with additional user preferences to rank and possibly refine the returned results (e.g. [82]).

The ever-increasing amounts of information that are available to a growing number of users complicates the too-many-answers problem and makes the need for personalization crucial. Instead of overwhelming users with all available data, a personalized query reports only the relevant to the users information according to their interests. Preferences can be used as a means to address this challenge. The goal of this thesis is the study of preferences in data management systems. To this end, we first introduce a context-dependent model for preferences and appropriate data structures and algorithms for managing contextual preferences. Then, we explore the integration of preferences to achieve personalized ranked retrieval (i) in keyword search in database systems and (ii) in publish/subscribe data delivery.

1.1 Thesis Contributions

The technical contributions of this thesis are along two axes. The first axis is centered on modeling issues. We propose a context-dependent model for preferences and focus on managing contextual preferences. We also address the problem of scoring database tuples using contextual preferences and provide a solution based on pre-computing representative rankings. The second axis is centered on integrating preferences in data management systems. In particular, we propose personalized keyword search in relational database systems and personalized delivery in publish/subscribe systems through user-defined preferences. As a side contribution of this thesis, we also consider an extensive survey of the use of preferences in databases.

Next, we summarize the contributions of this thesis.

A Survey on Preferences in Databases: Although several approaches that deal with preferences in databases have been proposed, a systematic study of these works is missing. In this thesis, we review the state of the art of these approaches and seek to understand the nature, representation and use of preferences from a database perspective. We classify approaches based on certain criteria. In particular, we present and compare existing approaches to preference representation followed by mechanisms for preference composition. Then, we study preferential query processing methods and, subsequently, we discuss preference learning approaches.

The results of this study are given in Chapter 2 and also appear in [108].

A Context-dependent Preference Model: We propose enhancing preferences with context-related information, since users may often have different preferences under differ-

ent circumstances. The novelty of this approach is that context is modeled using a set of multidimensional context parameters that take values from hierarchical domains. A contextual preference is defined as a triple (C, P, S), where: (i) C expresses conditions on the values of the context parameters, (ii) P expresses conditions on the values of the attributes of the database schema and (iii) S expresses a degree of interest. The meaning of such a contextual preference is that, under all circumstances specified by C, all database tuples that satisfy the conditions of P are assigned the indicated degree of interest S.

We also formulate the problem of context resolution as the problem of selecting appropriate preferences for personalizing a query based on context. We distinguish context resolution between: (i) the identification of all preferences with context that encompass the query context and (ii) the selection of the most appropriate preferences among these candidates. The first subproblem is resolved through the notion of cover partial order between contexts that relates contexts expressed at different levels of abstraction. To resolve the second subproblem, we consider appropriate distance metrics that capture similarity between contexts.

We introduce algorithms for context resolution that build upon two data structures, namely the *preference graph* and the *profile tree*. The preference graph explores the cover partial order of contexts to organize them in some form of lattice. A top-down traversal of the graph supports an incremental specialization of a given context, whereas a bottom-up traversal incremental relaxation. The profile tree offers a space-efficient representation of contexts by taking advantage of the co-occurrence of context values in preferences. It supports exact matches of contexts very efficiently through a single root-to-leaf traversal.

Finally, we evaluate our approach in terms of both usability and performance. Our usability experiments consider the overhead imposed to the users for specifying context-dependent preferences versus the quality of the personalization achieved. Our performance experiments focus on our context resolution algorithms that employ the proposed data structures to index preferences for improving response and storage overheads.

The proposed contextual preference model and the problem of context resolution are presented in Chapter 3 and also in [113, 111].

Scoring Database Tuples based on Contextual Preferences: We introduce a suite of techniques for quickly providing users with data of interest using our contextual preferences. In particular, we propose performing pre-processing steps to construct representative rankings of database tuples. To form such rankings, we create groups of similar preferences and produce a ranking for each group, considering as similar the preferences with similar contexts. To group preferences with similar contexts, we consider a contextual clustering method that exploits the hierarchical nature of context parameters. Our method can be applied to both quantitative and qualitative preferences. We also consider a complementary method for grouping preferences based on identifying those preferences that result in similar scores for all database tuples. This method takes advantage of the quantitative nature of preferences and groups together contextual preferences that have

similar predicates and scores. The method is based on a novel representation of preferences through a predicate bitmap table whose size depends on the desired precision for the resulting scoring. We present a number of experiments on both synthetic and real data sets.

The problem of contextual scoring database tuples is the subject of Chapter 4 and it has also been presented in [109, 110].

Personalized Keyword Search in Relational Database Systems: We propose personalizing keyword search through context-dependent preferences and provide a formal model for integrating preferential ranking with database keyword search. To the best of our knowledge, employing preferences to personalize keyword search is novel in this thesis.

Preferences express a user *choice* that holds under a specific *context*, where both context and choice are specified through keywords. We rank the results of a keyword query in an order compatible with the order expressed in the user choices for the query context. Furthermore, we combine the order of results as indicated by the user preferences with their relevance to the query. Besides preferences and relevance, given a constraint k on the number of results, we consider the set of the k results as a whole and seek to increase the overall value of this set to the users. Specifically, we aim at selecting the k most representative among the relevant and preferred results, i.e. these results that both cover different preferences and have different content.

We propose a number of algorithms for computing the top-k results. For generating results that follow the preference order, we rely on applying the winnow operator [32] on various levels to retrieve the most preferable choices at each level. Then, we introduce a *sharing-results* keyword query processing algorithm, that exploits the fact that the results of a keyword query are related with the results of its superset queries, to avoid redundant computations. Finally, we propose an algorithm that works in conjunction with the multi-level winnow and the sharing-results algorithm to compute the top-k representative results.

We evaluate both the efficiency and effectiveness of our approach. Our performance results show that the sharing-results algorithm significantly improves the execution time over the baseline. Furthermore, the overall overhead for preference expansion and diversification is reasonable. Our usability results indicate that users receive results more interesting to them when preferences are used.

The problem of personalizing keyword search is described in Chapter 5 and also in [106].

Preferential Publish/Subscribe: In a publish/subscribe system, users describe their interests through subscriptions and are notified whenever a published event matches any of their subscriptions. To control the rate of notifications received by the subscribers, we propose extending subscriptions to allow users to express the fact that some events are more important or relevant to them than others. To indicate priorities among subscriptions, we introduce *preferential subscriptions*. Events are ranked so that an event that

matches a highly preferred subscription is ranked higher than an event that matches a subscription with a lower preference.

Based on preferential subscriptions, we introduce a top-k variation of the publish/subscribe paradigm in which users receive only the matching events having the k highest ranks as opposed to all events matching their subscriptions. However, the top-k events are often very similar to each other. Besides pure accuracy achieved by matching the criteria set by the users, diversification, i.e. recommending items that differ from each other, has been shown to increase user satisfaction [136]. To this end, we adjust the top-k computation to take also into account the *diversity* of the delivered events. Since the generation of events is continuous, we also introduce a number of delivering policies for forwarding events.

As a proof-of-concept, we have implemented a prototype, termed PrefSIENA [5]. PrefSIENA extends SIENA [6], a popular publish/subscribe middleware system, with preferential subscriptions, delivering policies and diversity towards achieving top-k event delivery. We present a number of experimental results to assess the number of events delivered by PrefSIENA with respect to the original SIENA system, as well as, their rank and diversity. We also report on the overheads of supporting diversity-aware top-k delivery.

The problem of personalizing delivery in publish/subscribe systems is described in Chapter 6 and it has also been presented in [40, 41].

1.2 Thesis Outline

The rest of this thesis is structured as follows. In Chapter 2, we provide a framework for studying various approaches that deal with preferences in databases. We organize our study around the following axes: (*i*) preference representation, (*ii*) preference composition, (iii) preferential query processing and (*iv*) preference learning. In Chapter 3, we introduce a model for contextual preferences. We also formulate the context resolution problem as the problem of identifying the preferences with context that encompass the query context and selecting the most appropriate among them. In Chapter 4, we study the problem of scoring database tuples based on contextual preferences and provide a solution based on pre-computing representative rankings. In Chapter 5, we focus on personalizing keyword search. The proposed model exploits user preferences in ranking keyword results. In Chapter 6, we introduce a top-k variation of the publish/subscribe paradigm in which users receive only the k most interesting events as opposed to all events matching their subscriptions. Finally, Chapter 7 provides an overview of the results of this thesis and highlights open research challenges.

Chapter 2

Related Work on Preferences in Database Systems

In this chapter, we provide a framework for studying various approaches that deal with preferences in databases. We seek to understand the nature, representation and use of preferences from a database perspective. We consider ways in which the general notion of preference may be interpreted in a database system and we classify and evaluate approaches based on certain criteria. We identify the following main axes and organize our study around them:

Preference Representation: Preferences naturally come into different flavors and people may have a mix of different preferences, such as likes and dislikes, context-dependent and context-free, generic and specific and so forth. Capturing all possible preference types is a challenge. Different approaches to preference modeling focus on different aspects of preferences. Nevertheless, two main philosophies to preference modeling can be distinguished on the basis of how preferences are formulated: qualitative approaches, where preferences are expressed by comparing tuples ("I like westerns better than dramas") (e.g. [31]) and quantitative ones, where a preference for a specific tuple is expressed as a degree of interest in this tuple ("my interest in westerns is 0.8 and in dramas 0.4") (e.g. [11]). We categorize preference representation approaches based on the following dimensions:

1. *Formulation.* Preferences are formulated in a qualitative or quantitative way;

2. *Granularity.* Preferences can be expressed at different levels, i.e. for tuples, relations, relationships and attributes;

3. *Context.* Preferences can be context-free or hold under specific conditions, i.e. contextual preferences;

4. *Aspects.* Preferences may vary based on their intensity, elasticity, complexity and other aspects.

Preference Composition: Given a set of preferences for a set of objects in our domain, different composition methods or mechanisms to combine preferences can be applied to infer, combine or override preferences and finally, derive a ranking for the objects on the basis of how they match these preferences. We group preference composition mechanisms into the following categories:

1. *Qualitative composition.* These mechanisms combine preferences expressed for a set of tuples and order the tuples relatively to each other, i.e. in a qualitative way;

2. *Quantitative composition.* These mechanisms combine preferences expressed as scores over a set of tuples and assign final scores to these tuples that are ordered in a quantitative way;

3. *Heterogeneous composition.* These mechanisms are used to combine preferences of different granularity, for example, preferences for relationships between tuples with preferences for tuple attributes.

Preferential Query Processing: Preferential query processing methods exploit preferences to provide users with customized answers by changing the order and possibly the size of results (in the latter case, reporting only a portion of them, typically the top ranked ones). Several methods have been proposed by the database community focusing on integrating preferences into query processing. In this chapter, we present related work organized into the following topics:

1. *Expanding database queries.* These methods expand regular database queries with preferences by appropriately rewriting queries to incorporate preferences usually by adding selection conditions to the original query. In doing so, there are three fundamental steps: (*i*) determine which preferences are related to a specific query, (*ii*) identify how many and which of the related preferences should be integrated into the query and (*iii*) rewrite the original query to integrate the selected preferences and thus enable preferential query answering.

2. *Employing preference operators.* These methods employ special database operators to express preferences. There are two fundamental approaches to handling such

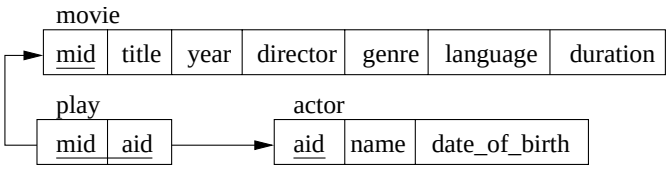


Figure 2.1: Movie database schema.

preference-related operators. One approach is to implement the operators inside the database engine. This involves developing appropriate evaluation methods as well as new query optimization algorithms. The other approach is to translate any preference operators in a query into existing relational algebra operators. This involves appropriately rewriting the original query so as to attain a cost-efficient query plan.

We shall also discuss methods for improving the performance of preferential query processing, for instance, by performing pre-processing steps off-line to construct rankings of database tuples based on preferences with the purpose of reducing the on-line query processing time.

Finally, since top-k processing is often an important component of preferential query processing, we shall also present a short taxonomy of related work and put work in this area in the context of preferential query processing.

Preference Learning: Learning and predicting preferences in an automatic way has attracted much research attention (e.g. [70, 35, 58]). Approaches to preference learning can be classified along various dimensions, such as: (*i*) the model learned, where there is a distinction between learning pairwise orderings of items (i.e. qualitative preferences) and learning a utility function (i.e. quantitative preferences), (*ii*) the input information type, where, in analogy to supervised and unsupervised learning, the learning algorithm may or may not use as input positive and/or negative examples, and (*iii*) the input classification dimension, where the learners may or may not use feedback from the users, which is usually in the sort of relevance judgment.

As a running example, we consider a simple database that maintains information about movies. Our movie database consists of three relations: movie, play, actor. Figure 2.1 depicts the schema of the movie database. We shall also use the instances of the relations that are shown in Figure 2.2.

The focus of this chapter is mainly on the representation, composition and use of preferences in databases and it is organized as follows. We present and compare existing approaches to preference representation (Section 2.1) followed by mechanisms for preference composition (Section 2.2). Subsequently, we study preference query processing methods (Section 2.3). Finally, we discuss preference learning approaches (Section 2.4).

	mid	*title*	*year*	*director*	*genre*	*language*	*duration*
t_1	m_1	Casablanca	1942	Curtiz	drama	english	102
t_2	m_2	Psycho	1960	Hitchcock	horror	english	109
t_3	m_3	Schindler's List	1993	Spielberg	drama	english	195

(a) movie relation

mid	*aid*
m_1	a_1
m_1	a_2
m_2	a_3
m_3	a_4

(b) play relation

aid	*name*	*date_of_birth*
a_1	H. Bogart	25-Dec-1899
a_2	I. Bergman	29-Aug-1915
a_3	A. Perkins	4-Apr-1932
a_4	L. Neeson	7-Jun-1952

(c) actor relation

Figure 2.2: Movie database instance example.

2.1 Preference Representation

In this section, we study and compare preference representation approaches based on the following dimensions: how preferences are formulated (*formulation* - Section 2.1.1), at what level they are expressed (*granularity* - Section 2.1.2), under which conditions they hold (*context* - Section 2.1.3) and what they express (*aspects* - Section 2.1.4).

2.1.1 Preference Formulation

Preferences can be expressed in a qualitative or a quantitative way. In the *qualitative approach*, the preferences between database tuples in the answer to a query are specified directly, typically using binary preference relations. Such relations provide an abstract, generic way to talk about a variety of concepts like priority, importance, relevance, timeliness, reliability etc. Preference relations can be defined using *logical formulas* [32] or special *preference constructors* [74] (which can be expressed using logical formulas). In the *quantitative approach*, preferences are expressed as numerical scores associated with database tuples. Scores can be assigned through *preference functions* (e.g. [11]) or as *degrees of interest* (e.g. [82, 113]) that map records (or record types) to scores. In general, a tuple t_i is preferred to a tuple t_j, if and only if, its score is higher than the score of t_j.

In the following, we shall use $R(A_1, \ldots, A_d)$ to denote a relational schema with d attributes A_i, $1 \leq i \leq d$, where each attribute A_i takes values from a domain $dom(A_i)$. Let A=$\{A_1, A_2, \ldots, A_d\}$ be the attribute set of R and $dom(A) = dom(A_1) \times \cdots \times dom(A_d)$ be its value domain. We use t to denote a tuple $(u_1, u_2, \ldots, u_d) \in dom(A)$ of R and r to denote an instance (i.e. tuple set) of R. Let $B \subseteq A$ be a subset of the attribute set, $t[B]$ stands for the projection of t on B. Finally, P denotes a preference.

Qualitative Preferences

In the qualitative approach, preferences are defined as binary relations between the tuples of the relational schema. Given a set S, a binary relation $\mathcal{B}$ over S is a subset of the Cartesian product $S \times S$. For a pair (a, b) which belongs to $\mathcal{B}$, we use the notation $a\ \mathcal{B}\ b$, whereas, for a pair (a, b) that does not belong to $\mathcal{B}$, we use the notation $\neg(a\ \mathcal{B}\ b)$.

A preference relation is defined as follows.

Definition 2.1. Let $R(A_1, \ldots, A_d)$ be a relational schema and $dom(A_i)$ be the domain of attribute A_i, $1 \leq i \leq d$. A preference relation $\succ_{PR}$ over R is a subset of $(dom(A_1) \times \cdots \times dom(A_d)) \times (dom(A_1) \times \cdots \times dom(A_d))$.

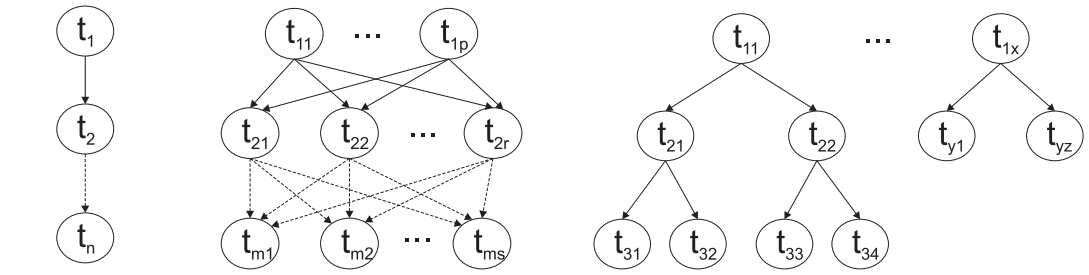
The interpretation of a preference relation $t_i \succ_{PR} t_j$ between two tuples t_i and t_j of R is that t_i is *preferred over* t_j under $\succ_{PR}$. We shall also say that t_i is *better than* t_j or that t_i *dominates* t_j under $\succ_{PR}$.

Next, we list several typical properties of binary relations that are useful in classifying preference relations. A binary relation $\mathcal{B}$ over a set S is called:

- reflexive, if, $\forall a \in S$, $a\ \mathcal{B}\ a$,
- irreflexive, if, $\forall a \in S$, $\neg\ (a\ \mathcal{B}\ a)$,
- symmetric, if, $\forall a, b \in S$, $a\ \mathcal{B}\ b \Rightarrow b\ \mathcal{B}\ a$,
- asymmetric, if, $\forall a, b \in S$, $a\ \mathcal{B}\ b \Rightarrow \neg(b\ \mathcal{B}\ a)$,
- antisymmetric, if, $\forall a, b \in S$, $(a\ \mathcal{B}\ b \wedge b\ \mathcal{B}\ a) \Rightarrow a = b$,
- transitive, if, $\forall a, b, c \in S$, $(a\ \mathcal{B}\ b \wedge b\ \mathcal{B}\ c) \Rightarrow a\ \mathcal{B}\ c$,
- negatively transitive (intransitive), if, $\forall a, b, c \in S$, $(\neg(a\ \mathcal{B}\ b) \wedge \neg(b\ \mathcal{B}\ c)) \Rightarrow \neg(a\ \mathcal{B}\ c)$,
- connective (strongly complete or total), if, $\forall a, b \in S$, $(a\ \mathcal{B}\ b) \vee (b\ \mathcal{B}\ a) \vee (a = b)$.

Note that the above properties are not independent. For instance, asymmetry implies irreflexivity, irreflexivity and transitivity imply asymmetry. In terms of a preference relation over a relational schema R, there is a subtle point regarding the set S over which the conditions of each property are tested. Typically, we should consider as S the set of all tuples $t = (u_1, u_2, \ldots, u_d)$, $u_i \in dom(A_i)$ of $R(A_1, A_2, \ldots A_d)$. This way, a preference relation $\succ_{PR}$ over R is asymmetric, if, $\forall t_i, t_j$ in any r of R, $t_i \succ_{PR} t_j \Rightarrow \neg(t_j \succ_{PR} t_i)$. However, in the presence of integrity constraints, we could apply the conditions only amongst tuples that all belong to a *valid instance* r of R, that is, to an instance r of R that does not violate any integrity constraints. Finally, one can consider a single (e.g. the current) instance of the database and test whether the tuples in the instance satisfy the conditions of the corresponding property.

Based on the subset of properties that hold for a preference relation $\succ_{PR}$, $\succ_{PR}$ is characterized as follows:



(a) Total order (b) Weak order (c) Strict partial order

Figure 2.3: Examples of preference graphs.

- A binary relation is a *preorder* or *quasi order*, if it is reflexive and transitive. If in addition, it is antisymmetric then it is a *partial order*.
- A binary relation is a *strict partial order* (or *irreflexive partial order*), if it is irreflexive, asymmetric and transitive. A preference relation $\succ_{PR}$ over a relational schema R is usually a strict partial order.
- A binary relation is a *total order*, if it is a strict partial order and it is also connective. If a preference relation $\succ_{PR}$ is a total order, any pair of tuples in any instance r of R is mutually comparable under $\succ_{PR}$.
- A binary relation is a *weak order*, if it is a negatively transitive strict partial order.

A preference relation among the tuples of an instance r of R can be represented through a directed graph, that we shall call *preference graph*. In the preference graph, there is one node for each tuple t in r and there is a directed edge from the node representing tuple t_i to the node representing tuple t_j, if and only if, $t_i \succ_{PR} t_j$. Some properties of the preference relation have a counterpart graph property.

If the preference relation is transitive, it is common to represent the transitive reduction of the relation. In particular, there is an edge from t_i to t_j, if and only if, $t_i \succ_{PR} t_j$ and $\nexists\, t_k$, such that, $t_i \succ_{PR} t_k$ and $t_k \succ_{PR} t_j$. The graph for a partially ordered set is also known as the Hasse diagram. In the following, we assume that preference relations are transitive and use the preference graph of their transitive reduction to represent them, unless stated otherwise. Examples of preference graphs for different types of preference relations are depicted in Figure 2.3. For instance, Figure 2.3a represents a preference relation among a set of tuples t_i, $1 \leq i \leq n$, such that, $t_j \succ t_{j+1}$, $1 \leq j \leq n-1$.

Besides the explicit listing of preference relations among tuples, a convenient way to express preferences between tuples is by using logical formulas to express the constraints that the two tuples must satisfy, so that, one is preferred over the other [32, 51].

Definition 2.2. Given a database instance r and two tuples $t_i, t_j \in r$, tuple t_i is preferred over tuple t_j based on a preference formula $PF(t_i, t_j)$, $t_i \succ_{PF} t_j$, if and only if, $PF(t_i, t_j)$ holds.

Preference formulas allow us to express choices between a large number of tuples in a single preference specification and in this respect, they can be considered set-oriented.

Example 1: Consider the relation *movie* provided in Figure 2.1 and its instance shown in Figure 2.2a. A user, say Addison, prefers a movie t from a movie t', if and only if, they are both of the same genre and t is longer than t' (preference P_1). Preference P_1 can be expressed as follows. Given two tuples t_i, $t_j \in r$, $t_i \succ_{P_1} t_j$, if and only if, $t_i[genre] = t_j[genre] \wedge t_i[duration] > t_j[duration]$. Consequently, t_3 is preferred over t_1 under P_1 in the given relation instance.

Based on the form of the formula PF, [32] makes a distinction between intrinsic and extrinsic preferences. *Intrinsic preferences* are specified based solely on the values of the two database tuples t_i and t_j that are being compared. *Extrinsic preferences* involve conditions that cannot be tested using only the values of the two tuples, such as, for example, the existence of other tuples in r, join conditions with tuples in other relations or comparisons of aggregate values. Using intrinsic preferences decouples the complexity of determining the preference order from the size of the database, since this involves only the two tuples being compared. It also makes the order insensitive to database updates.

In most cases, intrinsic preference formulas are restricted to first order quantifier-free formulas expressed in a disjunction of conjunctions normal form (DNF) of simple equality or rational-order constraints among the values of the two tuples. Specifically, a preference formula PF is a DNF of predicate conditions, $PF = (Cond_{1_1} \wedge \ldots \wedge Cond_{1_m}) \vee \ldots \vee (Cond_{p_1} \wedge \ldots \wedge Cond_{p_b})$, where each predicate condition $Cond_{x_y}$ has the form:

(i) $t_i[A_v]\ \theta_v\ t_j[A_v]$ or

(ii) $t_k[A_v]\ \theta_v\ co$,

where t_i, $t_j \in r$, $k \in \{i, j\}$, $A_v \in A$, $co \in dom(A_v)$ and $\theta_v \in \{=, \neq, <, >, \leq, \geq\}$ if A_v is a numerical attribute and $\theta_v \in \{=, \neq\}$ otherwise.

Example 2: Assume now that Addison prefers a drama movie that was recently produced or a horror movie with long duration (preference P_2). Preference P_2 can be expressed as follows. Given two tuples t_i, $t_j \in r$, $t_i \succ_{P_2} t_j$, if and only if, $(t_i[genre] = t_j[genre] \wedge t_i[genre] = \text{`}drama\text{'} \wedge t_i[year] > t_j[year]) \vee (t_i[genre] = t_j[genre] \wedge t_i[genre] = \text{`}horror\text{'} \wedge t_i[duration] > t_j[duration])$. In the given movie instance (Figure 2.2a), tuple t_3 is preferred over t_1 under P_2.

A formal language for formulating preference relations is proposed in [74]. The language has a number of base preference constructors, such as POS, NEG and $AROUND$, that are distinguished between numerical and non-numerical ones based on the domain type they are applicable to. For instance, P is a $POS(B, POS\text{-}set)$ preference if: $\forall\ t_i$, $t_j \in r$, $t_i \succ_P t_j$, if and only if, $t_i[B] \in POS\text{-}set \wedge t_j[B] \notin POS\text{-}set$, where $POS\text{-}set \subseteq dom(B)$. This means that a desired value of B is one that belongs to a finite set $POS\text{-}set$ of favorite values. For example, given the movie instance of Figure 2.2a and preference

$POS(\{director\}, \{Curtiz\})$, the most preferred tuple is the movie *Casablanca*. Similarly, a NEG preference defines that a desired value is one that does not belong to a finite set $NEG\text{-}set \subseteq dom(B)$ of dislikes. Preference constructors can be expressed using logical formulas.

Quantitative Preferences

In the quantitative approach, preferences are specified using functions that associate a numerical score with each database tuple. The score expresses the importance or degree of interest in the tuple.

Definition 2.3. Given an attribute set A, a function $f_P : dom(A) \rightarrow \mathbb{R}$ is a preference function that maps tuples to a score.

In general, t_i is preferred over t_j, that is, $t_i \succ_P t_j$ for a preference function f_P, if and only if, $f_P(t_i) > f_P(t_j)$. There are also approaches that consider the lowest scoring items as the highest ranked ones (e.g. [54]).

Example 3: Given a scoring function $f_{P_3}(t_i) = 0.001 \times t_i[duration]$ (preference P_3), the tuples t_1, t_2 and t_3 of the movie instance in Figure 2.2a have interest scores 0.102, 0.109 and 0.195 respectively. Therefore, t_3 is preferred over t_2 which in turn is preferred over t_1.

Generally, preference functions can operate on numerical and categorical attributes. A convenient way to describe preferences irrespective of the attribute domain is by specifying constraints that tuples must satisfy and assigning a preference score in these constraints. In particular, preferences can be expressed as scores (or *degrees of interest*) to selection conditions (e.g. [82, 113]). Preference scores belong to a predefined numerical domain. Usually, the score domain is in the range of real values between [0, 1], but any other integer or real range could be used.

Definition 2.4. Let $R(A_1, A_2, \ldots, A_d)$ be a relational schema and $dom(A_i)$ be the domain of attribute A_i, $1 \leq i \leq d$. A preference P on R is a pair ($Condition$, $Score$), where

1. $Condition$ is of the form $A_{i_1}\theta_{i_1}a_{i_1} \wedge A_{i_2}\theta_{i_2}a_{i_2} \wedge \ldots \wedge A_{i_k}\theta_{i_k}a_{i_k}$ and specifies a conjunction of simple selection conditions θ_{i_j} on the values $a_{i_j} \in dom(A_{i_j})$ of attributes A_{i_j}, $1 \leq i_j \leq d$, of R and

2. $Score$ belongs to a predefined numerical domain.

The meaning of such a preference is that all tuples from R that satisfy $Condition$ are assigned the indicated interest $Score$. $\theta \in \{=, <, >, \leq, \geq, \neq\}$ for numerical database attributes, while $\theta \in \{=, \neq\}$ for the categorical ones. Therefore, for a preference P, a tuple t_i is preferred to a tuple t_j, that is, $t_i \succ_P t_j$, if and only if, $Score_{t_i} > Score_{t_j}$.

Example 4: The preference ($movie.genre =$'$drama$', 0.9) shows a high interest in dramas (preference P_4), while the preference ($movie.year > 1990, 0.8$) shows interest in recent movies (preference P_5).

Analogously to qualitative preferences, quantitative preferences can be distinguished between intrinsic and extrinsic ones. Definition 4 describes intrinsic quantitative preferences, i.e. a preference score is assigned to a set of tuples based solely on the values of the tuple. In Section 2.1.2, we will see examples of extrinsic quantitative preferences based on conditions that cannot be tested using only the tuples for which a preference is expressed.

Equally Preferable and Incomparable Tuples

In an ideal world, preferences should exist for every object in a domain of interest. However, user preferences are typically incomplete. One can distinguish three cases related to preferences: equal preference, incompleteness and incomparability. Equal preference refers to two tuples being equally preferred. Incomparability represents that two tuples cannot in some fundamental sense be compared with each other and it is likely to arise when tuples combine together multiple features or when tuples are essentially very different (for example, a car and a bicycle). Incompleteness, on the other hand, represents a gap in our knowledge of user preferences. In general, it is not always possible to differentiate among the three.

Let us first look into specific types of a preference relation. In general, a preference relation $\succ_{PR}$ over a relational schema R induces an indifference relation $\sim_{PR}$, such that, $\forall\ t_i,\ t_j \in r,\ t_i \sim_{PR} t_j$, if and only if, $(\neg(t_i \succ_{PR} t_j) \wedge \neg(t_j \succ_{PR} t_i))$. Note that for a quantitative preference with scoring function f_P, $t_i \sim_f t_j \Leftrightarrow f_P(t_i) = f_P(t_j)$. If $\succ_{PR}$ is a total order, then the indifference relation $\sim_{PR}$ reduces to equality: $\forall\ t_i, t_j \in r,\ t_i \sim_{PR} t_j \Leftrightarrow t_i = t_j$. In this case, there is no gap in our knowledge. If $\succ_{PR}$ is a weak order, then indifference is an equivalence relation. A binary relation is an *equivalence relation* if it is reflexive, symmetric and transitive. Let $r/\sim_{PR}$ be the set of equivalence classes defined over each instance r by $\sim_{PR}$. This set is totally ordered by $\succ^*$, where $\succ^*$ is defined as: $\forall\ r_1, r_2 \in r/\sim_{PR}, r_1 \subseteq r, r_2 \subseteq r, r_1 \succ^* r_2$, if and only if, $t_i \succ_{PR} t_j$ for some $t_i \in r_1$ and $t_j \in r_2$. An example is shown in Fig. 2.4. In this case, all indifferent tuples in a class are in a sense equivalent to each other or equally preferred.

However, if the preference relation $\succ_{PR}$ is neither a total nor a weak order, the indifference relation $\sim_{PR}$ may not be transitive, thus it may not be an equivalence relation. To see this, consider the graph shown in Fig. 2.3c. The tuple t_{22} is indifferent to t_{21}, t_{21} is indifferent to t_{33}, but t_{22} is preferred over t_{33}. In this case, the indifference relation induced by the preference relation fails to capture the distinction between two tuples being incomparable versus being equally preferred. For example, the preference "I like drama and horror movies the same" should be interpreted differently from the preference "I cannot compare drama movies to horror movies". Next, we discuss how to make this distinction explicit for general types of a preference relation.

When $\succ_{PR}$ is a strict partial order, [75] propose to partition the indifference relation $\sim_{PR}$ into two parts: (i) a substitutable part and (ii) an alternative part. The substitutable part $\simeq_{PR}$ includes the tuples that can substitute each other in any preference expression and intuitively corresponds to tuples being equally preferred. Formally, the substitutable

relation $\simeq_{PR}$ is defined as: $\forall\ t_i, t_j \in r$, $t_i \simeq_{PR} t_j$, if and only if, (i) $t_i \simeq_{PR} t_j \Rightarrow t_i \sim_{PR} t_j$, (ii) $(\exists\ t_k \in r$, such that, $t_k \succ_{PR} t_i) \Rightarrow t_k \succ_{PR} t_j$, (iii) $(\exists\ t_k \in r$, such that, $t_i \succ_{PR} t_k) \Rightarrow t_j \succ_{PR} t_k$ and (iv) if we interchange t_i and t_j in (ii) and (iii), they still hold. The substitutable relation is reflexive, symmetric and transitive. Two indifferent tuples that are not substitutable are called alternatives. Alternatives tuples cannot substitute each other in preference expressions.

Example 6: In Fig. 2.3c, t_{31} can substitute t_{32}, and t_{33} can substitute t_{34}, whereas t_{21} and t_{22} are alternative tuples. As another example, consider the preference graph of Fig. 2.5, representing a strict partial order among six tuples. Here, tuples t_{11} and t_{12} are substitutable with each other and so are t_{21} with t_{22}, while for example, t_{11} and t_{21} are alternatives. Note that when $\succ_{PR}$ is a weak order, all tuples indifferent to each other (i.e., all tuples in the same equivalence class) are substitutable. For instance, tuples t_{21}, t_{22}, ..., t_{2r}, in Fig. 2.3b, are substitutable.

To differentiate between equally preferable and incomparable tuples, we could instead define a *strong indifference relation* [50], such that, $t_i \approx_{PR} t_j$, if and only if, $\forall\ t_k \in r$, $t_i \sim_{PR} t_k \Leftrightarrow t_j \sim_{PR} t_k$. For example, t_{11}, t_{12} and t_{21}, t_{22}, in Fig. 2.5, are strongly indifferent. Tuples related through strong indifference correspond to substitutable or equally preferred ones. Strong indifference $\approx_{PR}$ is an equivalence relation for any type of preference relation $\succ_{PR}$. Let $r/\approx_{PR}$ be the set of equivalence classes defined over each instance r by $\approx_{PR}$. Note that in this case, order $\succ^*$ defined over this set $r/\approx_{PR}$ of equivalence classes is not a total order.

An alternative way to differentiate between equally preferable and incomparable tuples is, instead of defining a preference order $\succ_{PR}$, to define another binary relation $\succsim_{PR}$ over R, such that, for two tuples t_i and t_j, $t_i \succsim t_j$ means that t_i is at least as good as or at least as preferable as t_j (e.g., [51]). The "at least as preferable" relation $\succsim$ is normally a preorder (i.e., reflexive and transitive). The binary relation $\succsim_{PR}$ induces a preference relation $\succ_{PR}$ over R, such that, $\forall\ t_i, t_j \in r$, $t_i \succ_{PR} t_j$, if and only if, $(t_i \succsim_{PR} t_j \wedge \neg(t_j \succsim_{PR} t_i))$. Note that this corresponds to the asymmetric part of $\succsim$. From the symmetric part of $\succsim$, we get an incomparability relation $||_{PR}$ as follows: $\forall\ t_i, t_j \in r$, $t_i\ ||_{PR}\ t_j$, if and only if, $(\neg(t_i \succsim_{PR} t_j) \wedge \neg(t_j \succsim_{PR} t_i))$. We also get an equally preferable relation $\simeq_{PR}$ as follows: $\forall\ t_i, t_j \in r$, $t_i \simeq_{PR} t_j$, if and only if, $((t_i \succsim_{PR} t_j) \wedge (t_j \succsim_{PR} t_i))$.

If $\succsim_{PR}$ is a preorder, $\succ_{PR}$ is a strict partial order. In addition, the equally preferable relation $\simeq_{PR}$ is transitive and thus, forms an equivalence relation. If $\succsim_{PR}$ is a preorder that is also antisymmetric, that is, if $\succsim_{PR}$ is a partial order, then $\simeq_{PR}$ is equality. Finally, if $\succsim_{PR}$ is a preorder that is also connective, $\succ_{PR}$ is a weak order and $||_{PR}$ is empty, that is, all tuples are comparable.

Qualitative vs. Quantitative Preferences

Qualitative preferences are described in a relative way through explicit tuple comparisons, while quantitative preferences are expressed in an absolute way directly on the desired tuples. In terms of expressive power, the qualitative specification of preferences is more

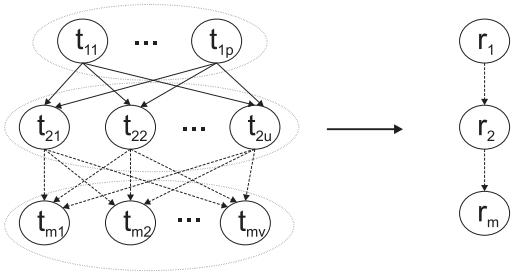


Figure 2.4: Tuples t_{kl} are related through the weak order $\succ_{PR}$. All tuples within each dotted oval are indifferent to each other and form an equivalence class. These equivalence classes r_k are totally ordered by $\succ^*$.

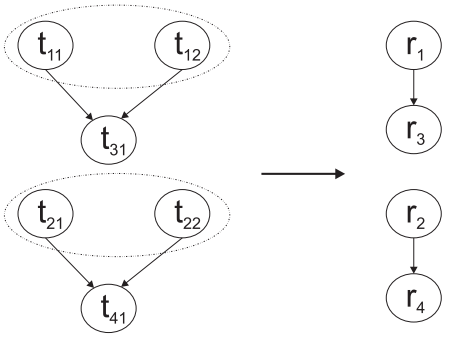


Figure 2.5: Examples of substitutable and strongly indifferent tuples.

general than the quantitative one, since not all preference relations can be expressed by scoring functions or through degrees of interest in conditions.

For example, take preference P_1 in Example 1: Addison prefers one movie tuple to another, if and only if, their genres are the same and the duration of the first is longer. Then in the example movie instance, t_3 is preferred over t_1 and t_2 cannot be compared with any of them. This preference cannot be expressed quantitatively, i.e., there is no scoring function that captures the previous preference. For the purpose of contradiction, assume that there is such a scoring function. Then, since there is no preference defined between any of the tuples t_1, t_3 and t_2, the score of t_2 should be equal to the scores of t_1 and t_3. But this implies that the scores of t_1 and t_3 are the same, which is not possible since t_3 is preferred over t_1. Note that $\succ_{P_1}$ is not a weak order. As another example, assume that Addison prefers to see recent movies over older ones, if the difference in the year of their production is at least two years, expressed by preference P_6: $\forall\ t_i, t_j \in r$, $t_i \succ_{P_6} t_j$, if and only if, $t_i[year] > t_j[year] + 2$. Similarly, P_6 cannot be expressed using a scoring function.

It has been shown that when the set over which the preference relation is defined is countable, a necessary and sufficient condition for a scoring function f_P, such that, $t_i \succ_P t_j \Leftrightarrow f_P(t_i) > f_P(t_j)$ to exist, is that $\succ_P$ is a weak order [50]. It is easy to see that if $\succ_P$ can be captured by a scoring function f_P, then it is a weak order. To prove sufficiency, assume that $\succ_P$ is a weak order. Then, the induced indifference relation $\sim_P$ is an equivalence relation. The trick for defining f_P is to assign scores to each tuple so

that (i) tuples that belong to the same equivalence class get the same score and (ii) for two tuples t_i and t_j that belong to two different equivalence classes r_1 and r_2 respectively, $f_P(t_i) > f_P(t_j)$, if and only if, $r_1 \succ^* r_2$.

Since using f_P implies that the produced preference relation is a weak order, we cannot find for preference relations that are not weak orders, a function f_P, such that, $t_i \succ_P t_j \Leftrightarrow f_P(t_i) > f_P(t_j)$. A less strict condition exists to preserve $\succ_P$ one way that is: $t_i \succ_P t_j \Rightarrow f_P(t_i) > f_P(t_j)$. For the example preference P_1, we could for example define f_P as, $f_P(t_1) = 3$, $f_P(t_2) = 2$ and $f_P(t_3) = 1$.

In particular, it has been shown that, when the set over which the preference relation is defined is countable, a necessary and sufficient condition for a scoring function f_P, such that, $t_i \succ_P t_j \Rightarrow f_P(t_i) > f_P(t_j)$ and $t_i \approx_P t_j \Leftrightarrow f_P(t_i) = f_P(t_j)$ to exist, is that $\succ_P$ is acyclic [50]. Relation $\succ_P$ is *acyclic* if we never have $t_1 \succ_P t_2 \succ_P \ldots \succ_P t_m \succ_P t_1$, for finite m. The idea here is to use the equivalence classes induced by strong indifference for assigning scores. In this case, the order $\succ^*$ among the equivalence classes is not a total order.

However, qualitative preferences return the most preferred tuples without distinguishing how much better one tuple is compared to another. For example, they cannot distinguish preferences, such as "I like comedies very much" vs. "I like dramas a little" and they cannot capture different feelings, such as dislike in dramas.

2.1.2 Preference Granularity

Irrespective of their (qualitative or quantitative) formulation, preferences may be expressed at different levels of granularity.

Tuple preferences are expressed directly over database tuples. Typically, they are formulated over the tuples of a single relation based on the attributes of this relation [32, 74, 113]. In the previous subsection, we examined several examples of tuple preferences.

[81] allows preferences for tuples in a relation R to be formulated based on values of attributes in different relations that join to R. In essence, Definition 2.4 can be generalized as follows: a preference P on a relation R is a pair $(Condition, Score)$, where $Condition$ is a conjunction of atomic selections involving a set of attributes B in the database and atomic joins transitively connecting these attributes to R. This is an example of a definition of an extrinsic quantitative preferences.

Example 7: Consider the database schema in Figure 2.1. A preference for movies can be specified using attributes of movies, such as the year of a movie, but also attributes that are implicitly associated with movies and are stored in other relations, such as the names of the actors starring in a movie. For instance, a preference for movies with Julia Roberts could be stated as follows (preference P_7):

movie : ($movie.mid = play.mid$ and $play.aid = actor.aid$
and $actor.name =$ '*Julia Roberts*', 0.8).

Set preferences are expressed based not only on the individual tuple properties, but

also on the properties of a group of tuples as a whole. Typically, a set preference can be expressed on the desired quantity of certain objects in a set. For example, Addison wants to watch three movies one of which has to be a comedy. Or she may want to see three movies of the same director.

[135] represents set preferences in a qualitative way. For describing sets of tuples, the concept of set profile is employed. A set profile is defined as a collection of features that is mapped to a tuple. These features are generated with an aggregate query with *min*, *max*, *sum*, *count*, *avg*. Then, a set preference is reduced to a tuple preference over set profiles of the same cardinality.

Attribute preferences are expressed on attributes and set priorities between the attributes of tuples.

Attribute preferences have different interpretations over the result of a database query in different approaches [51, 92]. For example, they can be used to set priorities among tuple preferences that are expressed over the values in the corresponding attributes (e.g. [51]).

Example 8: Addison prefers the director Steven Spielberg to David Lynch (preference P_d) and adventures to dramas (preference P_g), but she considers the director more important than the movie genre. This attribute preference can be expressed as $P_d \succ P_g$.

Alternatively, attribute preferences can express priorities among the attributes to be displayed in the result of a query. For instance, quantitative attribute preferences, called $\pi - preferences$, are discussed in the framework of [92]. A $\pi - preference$ is expressed by assigning an interest score to relation attributes.

Example 9: Assume Addison is interested only in the title, genre and language of a movie. Then, she may express the following preferences:
$P_{\pi_1} = (\{title, genre, language\}, 1)$,
$P_{\pi_2} = (\{year, director, duration\}, 0.2)$.

Relationship preferences are expressed on a relationship between two types of objects (called *generic*) or two particular objects (called *instance*). A generic relationship preference shows interest in a particular type of relationship. For example, one may consider significant the relationship "a director has directed movies". On the other hand, one may think that the relationship "Julia Roberts acted in Ocean's Twelve" is not important. This is an instance relationship preference.

A framework that supports both tuple and relationship preferences for different relations of a database schema is presented in [82]. User preferences are stored as degrees of interest in atomic query elements that can be individual selection or join conditions (called *selection* and *join preferences* respectively). Join preferences show a user interest in tuples from a relation R that join to tuples in different relations for which preferences do exist.

Example 10: Addison likes actress Julia Roberts and expresses a selection preference:

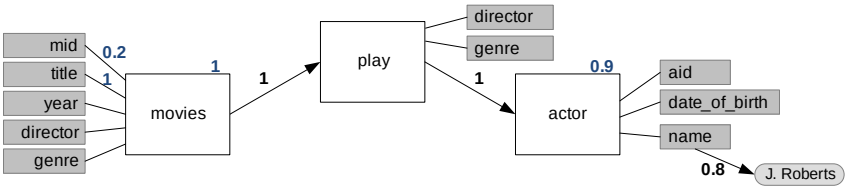


Figure 2.6: Personalization graph.

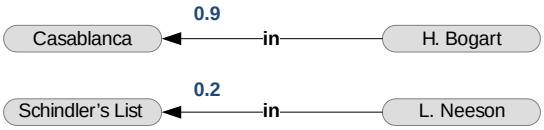


Figure 2.7: Example instance relationship preferences.

actor : (*actor.name* ='*Julia Roberts*', 0.8). Moreover, she has preferences expressed over the joins between the relations of the database schema (Figure 2.1). In particular, she considers the actor of a movie very important and hence she has the following join preferences: *movie* : $(movie.mid = play.mid, 1)$ and *play* : $(play.aid = actor.aid, 1)$. These join preferences essentially connect movies to their actors and show that movie preferences are also shaped by preferences for their actors.

In order to map simple preferences for tuples and relationships over a database, [82] introduced the concept of a *personalization graph* $G(V, E)$, a directed graph that is an extension of the database schema graph. The nodes in V are (i) relation nodes, one for each relation of the schema, (ii) attribute nodes, one for each attribute of each relation of the schema and (iii) value nodes, one for each value for which a user expresses a preference. Edges in E are selection edges, representing a possible selection condition from an attribute to a value node, and join edges, representing a join between relations. Tuple preferences map to selection edges. Join edges capture generic relationship preferences. Figure 2.6 illustrates a personalization graph that captures the preferences given in Example 10. A personalization graph could also capture attribute and relation preferences, expressed in a similar way as preferences in [82], i.e. as degrees of interest. Figure 2.6 illustrates example preferences for the relations and attributes in our movie database in different color.

In the same philosophy as above, one could imagine an instance-based personalization graph, like the one depicted in Figure 2.7, where nodes correspond to tuples and edges correspond to relationships between tuples.

Relation preferences are expressed on a type or class of objects. For example, "I am interested in directors, but not in producers".

To the best of our knowledge, relation preferences are not found in the literature so far. However, it is noteworthy that the granularity at which preferences can be expressed is determined to a great extent by the database schema. Hence, it is possible to express the same preference at different granularities in different schemas for the same data. For

example, consider that instead of the movie relation depicted in Figure 2.1, we have the following relations:

$movie(mid, title, year, language, duration, did)$,

$director(did, name)$, $genre(mid, genre)$

Then, the preference for the director over the genre expressed as an attribute preference in Example 8, would be instead formulated as a relationship preference.

2.1.3 Context

Context is a general term used to capture any information that can be used to characterize the situations of an entity [39]. Changing the behavior of an application taking into account the current user context has been extensively studied in the context-aware computing community, where different definitions of context have been proposed (e.g. [23, 103]). A generic definition is the following [39]:

"Context is any information that can be used to characterize the situation of an entity. An entity is a person, place or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."

It is noteworthy that under this general definition user preferences can be also considered part of the user context because they characterize the situation of a user and are used to adapt a system's answers to user queries. Here, we study how the context determines when and how user preferences should be considered and we will consequently see how it becomes part of preferences formulation. With these observations in mind, we present here our own definition of context:

Definition 2.5. Context is any external information that can be used to characterize the situation of a user or any internal information that can be used to characterize the data per se.

Preferences may hold unconditionally, under all circumstances. These are called *context-free preferences*. The examples we studied in the previous subsections are such. There are also preferences that hold under specific circumstances or in a particular context. In the general form, a contextual preference consists of two parts: a context part and a preference part. Formally:

Definition 2.6. A contextual preference CP is a pair $(\mathcal{C}, P)$, where $\mathcal{C}$ defines the context and P defines the preference.

The context part specifies the conditions under which the preference part holds. Based on our definition of context, the context can be dictated by our data or it can be ephemeral, volatile and external to the database. On the basis of this distinction, contextual preferences fall into two categories: *internal contextual preferences* and *external contextual preferences*. For example, an internal contextual preference could be that for animations the year of release should be after 2000. On the other hand, a preference for thrillers after

Table 2.1: Preference representation approaches w.r.t. context and preference formulation.

		Formulation	
		Qualitative	*Quantitative*
Context	*Internal*	[10, 32]	−−
	External	[60]	[112, 113, 92, 119, 120]

midnight is an external contextual preference and, in particular, it is a time-dependent one.

Contextual preference models have been introduced following both the qualitative [10, 60, 32] and the quantitative [112, 113, 92] approach for describing the preference part. Furthermore, knowledge-based contextual preferences have been proposed [119]. Table 2.1 presents a taxonomy of contextual preference models with regards to context and preference formulation.

In the following subsections, we will focus on the context part of these preference representations.

Internal Contextual Preferences

In an internal contextual preference $(\mathcal{C}, P)$, the context part $\mathcal{C}$ is modeled using key-value pairs, or tuples, and includes attributes that are part of the database schema (e.g. [10]). Formally:

Definition 2.7. Given an instance r of a relation schema $R(A_1, A_2, \ldots, A_d)$, an internal context $\mathcal{C}$ is $\wedge_{j \in l}(A_j = a_j)$, $l \subseteq \{1, \ldots, d\}$ and $a_j \in dom(A_j)$.

[10] defines contextual qualitative preferences to rank database tuples. The preference part is called *choice*. Given an instance r of a relation schema $R(A_1, A_2, \ldots, A_d)$, an internal contextual preference CP is of the form $\{A_i = a_{i_1} \succ A_i = a_{i_2} \mid \mathcal{C}\}$, where a_{i_1}, $a_{i_2} \in dom(A_i)$ and $\mathcal{C}$ is $\wedge_{j \in l}(A_j = a_j)$, $l \subseteq \{1, \ldots, d\}$ and $a_j \in dom(A_j)$. The meaning of such a contextual preference is the following: all tuples $t \in r$, such that, $t[A_i] = a_{i_1}$ and $t[A_j] = a_j$, $\forall j \in l$, are preferred to tuples $t' \in r$ for which $t'[A_i] = a_{i_2}$ and $t'[A_j] = a_j$, $\forall j \in l$.

Example 11: Consider the following internal contextual preference: $\{director = \text{`}Spielberg\text{'} \succ director = \text{`}Curtiz\text{'} \mid genre = \text{`}drama\text{'}\}$. Then, tuple $t_3$ of the movie instance shown in Figure 2.2a is preferred to tuple $t_1$ in the context of dramas, while the preference $\{director = \text{`}Curtiz\text{'} \succ director = \text{`}Spielberg\text{'} \mid genre = \text{`}drama\text{'} \wedge language = \text{`}English\text{'}\}$ defines that t_1 is preferred to t_3 in the context of English dramas.

Similarly, [32] proposes the concept of conditional preferences, that is, preferences that depend on the presence of specific attribute values. The proposed model permits disjunctions of such preferences. For instance, consider that Addison prefers movies directed

by Hitchcock if they are horrors and movies directed by Curtiz if they are dramas, in the sense that, any Hitchcock movie is preferred over any Curtiz movie for horrors, while the opposite holds for dramas. This preference can be expressed as follows. Given two tuples t_i, $t_j \in r$, $t_i \succ_P t_j$, if and only if, $(t_i[genre] = t_j[genre] \wedge t_i[genre] = \text{`}drama\text{'} \wedge t_i[director] = \text{`}Hitchcock\text{'} \wedge t_j[director] = \text{`}Curtiz\text{'}) \vee (t_i[genre] = t_j[genre] \wedge t_i[genre] = \text{`}horror\text{'} \wedge t_i[director] = \text{`}Curtiz\text{'} \wedge t_j[director] = \text{`}Hitchcock\text{'})$.

External Contextual Preferences

In an external contextual preference $(\mathcal{C}, P)$, the context part $\mathcal{C}$ is described as a situation outside the database (e.g. [112, 113, 60, 119, 92]). Common types of external context include the *computing context* (e.g. network connectivity, nearby resources), the *user context* (e.g. profile, location), the *physical context* (e.g. noise levels, temperature) and *time* [30]. [18] reviews recent evolutions of context modeling.

To model external context, we use a finite set of special-purpose attributes, called context parameters. Formally:

Definition 2.8. Given a set of context parameters C_1, ..., C_n with domains $dom(C_1)$, ..., $dom(C_n)$ respectively, an external context $\mathcal{C}$ is a n-tuple of the form $(c_1, \ldots, c_n)$, where $c_i \in dom(C_i)$.

In our work [112, 113], we express external contextual preferences in a quantitative way, where context parameters take values from hierarchical domains. The model used in [112] for defining preferences includes only a single context parameter. Interest scores of preferences involving more than one parameter are computed by a simple weighted sum of the scores of the preferences expressed with single parameters. This approach is extended in [113], where contextual preferences involve more than one context parameter and may hold for a set of external contexts. More details about our context model are given in Chapter 3. Contextual preferences of the same form and with a similar context model are also defined in [92].

Example 12: Assume the context parameters *accompanying_people* and *time_period* and also that Addison enjoys seeing comedies when accompanied with friends during holidays or weekends. Such a contextual preference assigns a high interest score to comedies under the external contexts $(friends, holidays)$ and $(friends, weekends)$.

External contexts, termed situations, are also discussed in [60]. Each context has an identifier *cid* and consists of one timestamp, one location and other influences, such as physical state, current emotion, weather conditions or accompanying people. External contexts are uniquely linked through an N:M relationship with preferences expressed using the qualitative approach introduced in [74]. Therefore, an external contextual preference can be considered as a tuple (cid, pid), expressing that the qualitative preference *pid* holds in the context *cid*.

Finally, a knowledge-based contextual preference model is proposed in [119]. In this approach, a variant of description logics is explored to model user preferences. Contextual

preferences, called *preference rules*, have again the form $(\mathcal{C}, P)$, where both context and preference are description logics concept expressions. To rank the answers of a query, preference rules are enhanced with numerical scores [120]. The approach results in an explanatory scoring method for documents, which is related to traditional, non context-aware, information retrieval.

2.1.4 Preference Aspects

Preferences naturally come into different flavors and can express different opinions and desires. For example, a preference may express like (e.g. "I like going to movies") or dislike (e.g. "I do not like long movies"). It may capture a general rule (e.g. "I like all comedies") or a finer-grained taste (e.g. "I like comedies released after 2000 by American directors") and so forth.

In what follows, we consider a set of preference-related concepts and we describe several types of preferences with the purpose of further understanding the expressivity of existing preference modeling works. Of course, we do not aim at delving into philosophical questions of what preferences may express.

Intensity shows the degree of desire expressed in a preference. Synonym concepts are priority, significance and importance. Intensity answers the question of "how strong a preference is". Preferences can be loosely characterized as *strong*, *weak* or *moderate*. As we have discussed back in Section 2.1.1, quantitative approaches capture intensity in the score or degree of interest attached to a preference. For example, the preference $(movie.genre = \text{`}drama', 0.9)$ is a strong preference compared to $(movie.genre = \text{`}animation', 0.3)$, which can be considered a weak preference. On the other hand, qualitative approaches can show intensity only in an abstract way by the virtue of comparison.

Necessity corresponds to the satisfaction of a preference and answers to the question "should the preference be met". A preference may be *hard or mandatory* if it must be absolutely satisfied. Note that the conditions in a query are considered by default hard in the traditional boolean database query model. Moreover, a user may also have preferences that are hard constraints. For example, the preference "If I am with friends, I do not want to see a drama movie" is a hard constraint. *Soft or optional* preferences are desired and may be satisfied. For instance, a preference for director W. Allen is optional, since this director may not be the only criterion for selecting a movie. Necessity may be related to the intensity of a preference. For example, in [82], necessity is captured through the intensity of a preference, as a degree of interest equal to 1 shows mandatory preference whereas values less than 1 show optional preferences. In the general case, necessity may not be expressed through a preference's intensity. For example, a *veto* holds over and blocks other preferences that may be conflicting with it. In [11], a veto for a particular value is captured by the symbol $\natural$.

Feeling answers to the question "how one feels about something". Preferences may be *positive*, i.e. expressing like, *negative*, expressing dislike or *indifferent* if they convey no

particular taste. Quantitative approaches can capture more easily negative preferences in the score. For example, a degree of interest in the range [-1, 0) shows dislike [80], while a degree equal to 0 indicates indifference. Indifference is also explicitly captured in [11] with the symbol $\perp$. Since qualitative approaches are based on comparisons, comparisons cannot explicitly distinguish between something that is less desired vs. something not liked.

Positive and negative preferences can also be captured through the preference constructors proposed in [74]. Examples of such constructors are the POS and NEG preferences described in Section 2.1.1.

Complexity describes the degree of detail expressed in a preference and answers to the question "how specific a preference is". A preference may be *generic* or *simple* if it refers to a single relationship or attribute of the objects of interest. For example, a qualitative preference that compares a pair of entities based on a single attribute is a simple preference. A *compound* preference jointly expresses a combination of simple preferences to be concurrently met. For example, a preference for "comedies directed by Woody Allen after 2000" is a compound preference expressing a very specific interest in a subclass of comedies [32, 81].

Attitude indicates whether a preference is associated with the existence (*presence* preference) or absence of certain values or relationships (*absence* preference). For instance, a preference for a movie being a comedy is a presence preference whereas a preference for movies without violence is an absence one. The model described in [80] allows capturing both presence and absence preferences associated with different degrees of interest (i.e. of different intensities). Also, a veto expresses a prohibition on the presence of a specific set of values in the elements of the answer to a query [11, 32]. In our terminology, a veto is a hard absence preference.

Elasticity indicates how strict a preference is. An *exact* preference is either satisfied exactly or not at all. Given the mutual independence of categorical values, preferences for categorical values are typically considered exact. For instance, a preference for adventures is an exact one if no way is provided in order to specify which "adventure-like" movies would be also acceptable. *Elastic* preferences may be satisfied as closely as possible and are usually associated with numerical values. For example, a preference for the duration of a movie is more naturally expressed as an elastic one, such as "I like movies around 2h". One could imagine using a similarity function to capture elastic preferences on categorical values as well.

Qualitative approaches capture exact preferences (e.g. [32]). On the other hand, scoring or preference functions express elastic preferences for numeric data [11]. [74] captures elastic preferences for numerical attributes using special order and distance operators. For example, the $AROUND(B, z)$ preference constructor denotes that the desired tuples have value z for B. If this is infeasible, the tuples with values with the smallest distance from z are the preferable ones. For instance, under the preference $AROUND(\{duration\}, 180)$, the most preferable movie in the database in Figure 3.2 is *Schindler's List*, since it has

the closest duration to the desired one.

[80] captures elastic preferences in the form of the degree of interest. If the degree is a constant value, like in $(movie.genre = `animation', 0.3)$, then it expresses a preference for the exact value specified. The degree of interest can also be a function over the domain of an attribute values and, in this case, it captures an elastic preference not for a single value but over a range of values of varying intensity.

2.1.5 A Summary of Preference Representations

To synopsize, preference representation approaches can be categorized based on the following dimensions: how preferences are formulated (*formulation*), at what level they are expressed (*granularity*), under which conditions they hold (*context*) and what they express (*aspects*). Preferences can be expressed in a qualitative or in a quantitative way over tuples, relations, attributes or relationships. Context-free preferences hold under all possible contexts whereas contextual preferences can hold in a specific context that can be internal or external to the database. Finally, preferences can be characterized based on various aspects, such as intensity, feeling and complexity. Figure 2.8 summarizes the various options in each dimension.

Table 2.2 compares different approaches for preference representation based on the formulation, granularity and context dimensions. Table 2.3 drills down to preference types and compares approaches based on what preference aspects are captured at each granularity. [84] opened the road to preference modeling in databases, but they did not specify a formal definition of the proposed language. We observe that most works have focused on tuple preferences, while preferences at the level of relation have not been considered yet. For example, we could define how books in general are preferred by a user compared to movies. A second observation is that current representation approaches are pure qualitative or quantitative. Only [74] covers preferences that could be formulated in a mixed way. For example, one could imagine a scenario where we have two users, one's preferences are captured in a qualitative way and the other's in a quantitative way. The objective is to return results that capture both users' preferences.

2.2 Preference Composition

Given a set of preferences, there are different composition methods or mechanisms to combine preferences and determine the ultimate ranking of a set of affected tuples. In this section, we distinguish such methods based on two criteria: *user attitude* and *tuple ranking*. Let us assume that we have two preferences P_x and P_y . Then, we distinguish between the following attitudes:

- *Overriding attitude*: Preference P_x overriding P_y means that P_y is applicable only if P_x does not apply;

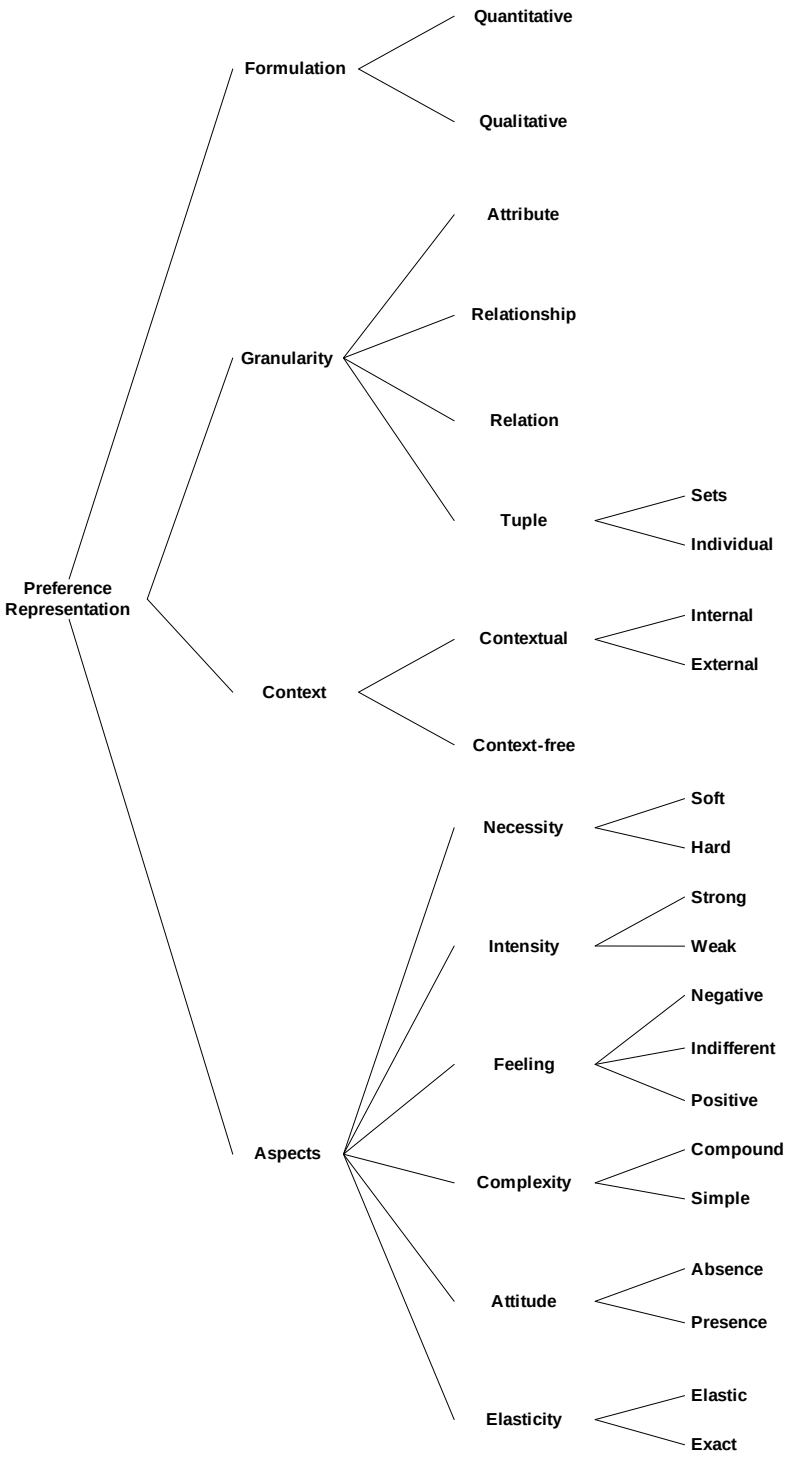


Figure 2.8: A taxonomy of preference representations.

Table 2.2: Preference representation approaches w.r.t. preference formulation, granularity and context.

	Formulation		**Granularity**				**Context**		
	Qualitative	Quantitative	Tuple	Relation	Attribute	Relationship	Context-free	Internal	External
[84]	✓		✓				✓		
[135]	✓		sets				✓		
[31, 32]	✓		✓				✓	✓	
[74]	✓	✓	✓				✓		
[11]		✓	✓				✓		
[82, 80]		✓	✓			✓	✓		
[10]	✓		✓					✓	
[112, 113]		✓	✓				✓		✓
[60]	✓		✓						✓
[119, 120]		✓	✓						✓
[51]	✓		✓		✓		✓		
[92]		✓	✓		✓				✓

Table 2.3: Preference representation approaches w.r.t. preference aspects (T=tuple, C=relation, A=attribute, R=relationship).

	Aspects												
	Intensity		Necessity		Feeling			Complexity		Attitude		Elasticity	
	strong	*weak*	*hard*	*soft*	*positive*	*negative*	*indifferent*	*simple*	*compound*	*presence*	*absence*	*exact*	*elastic*
[84]	T	T	–	T	T	–	–	T	T	T	–	T	–
[31, 32], [135]	T	T	–	T	T	–	T	T	T	T	T	T	–
[74]	T	T	–	T	T	T	–	T	T	T	T	T	T
[11]	T	T	–	T	T	–	T	T	T	T	T	T	T
[82, 80, 81]	T	T	TR	TR	T	T	T	TR	TR	T	T	T	T
[10]	T	T	–	T	T	–	–	T	T	T	–	T	–
[112, 113]	T	T	–	T	T	–	–	T	T	T	T	T	–
[60]	T	T	–	T	T	T	–	T	T	T	T	T	T
[119, 120]	T	T	–	T	T	–	–	T	T	T	T	T	–
[51]	TA	TA	A	T	TA	–	TA	T	T	TA	–	TA	–
[92]	TA	TA	A	TA	TA	–	–	TA	TA	TA	T	TA	–

- *Dominant attitude*: The most or least important preference determines the tuple ranking;
- *Combinatory attitude*: Both P_x and P_y contribute to the tuple ranking.

Preference composition may order a pair of tuples in a qualitative or in a quantitative way. *Qualitative composition* mechanisms order the tuples relatively to each other, whereas *quantitative composition* mechanisms combine preferences expressed as scores over a set of tuples and assign final scores to the tuples, which are thus ordered in a quantitative way.

We present different ways of combining preferences over tuples (Section 2.2.1). Apart from mechanisms for composing tuple preferences, there are also composition mechanisms for preferences of different granularity, for example, preferences for tuples with preferences for attributes. Hence, we also discuss cases of *heterogeneous composition* (Section 2.2.2).

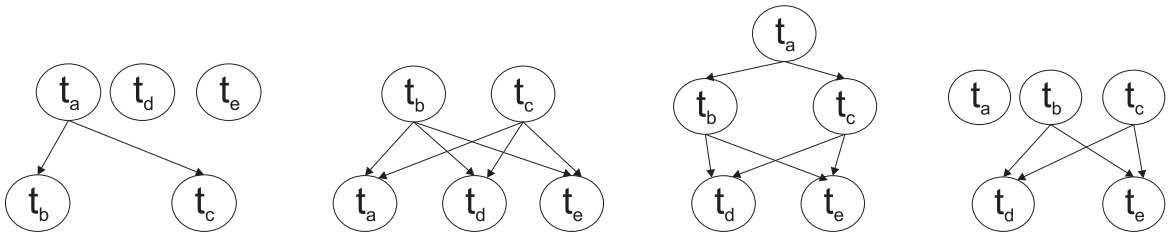
2.2.1 Combining Preferences for Tuples

As we discussed in Section 2.1.1, qualitative preference representation approaches are more general than quantitative ones, in the sense that we cannot in general find a scoring function f_P, such that $t_i \succ_P t_j \Leftrightarrow f_P(t_i) > f_P(t_j)$. On the other hand, for each f_P, there is a preference relation $\succ_P$, such that $f_P(t_i) > f_P(t_j) \Rightarrow t_i \succ_P t_j$. Based on this observation, every composition mechanism defined over preference relations can be also applied to preferences defined using functions (or degrees of interest) and leads to a qualitative tuple ranking. Consequently, in what follows, we describe different composition mechanisms for preference relations (prioritized composition, lexicographical, pareto, intersection, union, difference and transitive closure), which are also applicable to numerical preferences (although in our discussion, we will mention only preference relations for brevity). We also describe numerical (i.e. quantitative) composition that can be applied to quantitative preferences (and qualitative mapped to quantitative ones). Without loss of generality, in the following, we assume composition of two preferences P_x and P_y; generalizing to $n > 2$ preferences is straightforward.

In general, composition can be distinguished between (i) composition of preference relations over the same schema R, in which case the composed preference relation is defined over R, and (ii) composition of preference relations over different schemas R and R', in which case the composed preference relation is defined over the Cartesian product $R \times R'$.

Qualitative Composition

Prioritized Preference Composition. Given the preferences P_x and P_y, in a prioritized preference $P_x \& P_y$, P_x is considered more important than P_y. Formally:



(a) P_x (b) P_y (c) $P_x \& P_y$ (d) $P_x \otimes P_y$

Figure 2.9: Example of prioritized and pareto composition.

Definition 2.9. Let P_x and P_y be two preference relations defined over the same relational schema R. The prioritized preference composition relation $\succ_{P_x \& P_y}$ is defined over R, such that, $\forall\ t_i$, t_j of R, $t_i \succ_{P_x \& P_y} t_j$, if and only if, $(t_i \succ_{P_x} t_j) \vee (t_i \sim_{P_x} t_j \wedge t_i \succ_{P_y} t_j)$.

The intuitive meaning of prioritized composition is: use preference P_y only if P_x is not applicable. An example is shown in Figure 2.9, where the preference graph in Figure 2.9c represents the preference resulting from the prioritized composition of the preferences P_x and P_y represented by the preference graphs shown in Figure 2.9a and Fig 2.9b respectively. Using logical formulas, we consider the following example.

Example 13: Addison prefers drama movies over horror movies (preference P_8) and long movies over short ones (preference P_9). Preference P_8 can be defined using logical formulas as: $t_i \succ_{P_8} t_j$, if and only if, $t_i[genre] =$ '$drama$' $\wedge\ t_j[genre] =$ '$horror$'. Similarly, P_9 can be expressed as: $t_i \succ_{P_9} t_j$, if and only if, $t_i[duration] > t_j[dura\text{- }tion]$. Then, the prioritized preference $P_8 \& P_9$ can be defined as: $t_i \succ_{P_8 \& P_9} t_j$, if and only if, $(t_i[genre] =$ '$drama$' $\wedge\ t_j[genre] =$ '$horror$') $\vee\ (t_i[genre] \neq$ '$drama$' $\wedge\ t_i[duration] > t_j[duration])$ $\vee\ (t_j[genre] \neq$ '$horror$' $\wedge\ t_i[duration] > t_j[duration])$. Now, for the movie relation of Figure 2.2a, under $P_8 \& P_9$, t_3 is preferred over t_1 which in turn is preferred over t_2.

Prioritized composition may also be applied among preferences defined over different relational schemas. In this case, prioritized composition is called *lexicographical composition*. Formally:

Definition 2.10. Let P_x and P_y be two preference relations defined over the relational schemas R and R' with attribute domains $dom(A)$ and $dom(A')$ respectively. The lexicographical preference composition relation $\succ_{P_x \& P_y}$ defined over the Cartesian product $R \times R'$ is a subset of $dom(A) \times dom(A')$, such that $(t_i, t'_i) \succ_{P_x \& P_y} (t_j, t'_j)$, if and only if, $(t_i \succ_{P_x} t_j) \vee (t_i \sim_{P_x} t_j \wedge t'_i \succ_{P_y} t'_j)$, where t_i, t_j are tuples of R and t'_i, t'_j tuples of R'.

It has been shown that total and weak orders are preserved by prioritized and lexicographical composition, whereas, strict partial order is not [32]. Prioritized composition is easily extended for the "at least as preferable" relation $\succsim_{PR}$. In particular, for lexicographical composition [51]:
(i) $(t_i, t'_i) \succ_{P_x \& P_y} (t_j, t'_j)$, if and only if, $(t_i \succ_{P_x} t_j) \vee (t_i \simeq_{P_x} t_j \wedge t'_i \succ_{P_y} t'_j)$, (ii) $(t_i, t'_i) \simeq_{P_x \& P_y} (t_j, t'_j)$, if and only if, $t_i \simeq_{P_x} t_j \wedge t'_i \simeq_{P_y} t'_j$ and (iii) $(t_i, t'_i)\ ||_{P_x \& P_y} (t_j, t'_j)$ otherwise.

For P_x and P_y described with the help of query conditions over the same relation R, i.e. of the form $R : (q_x, d_x)$ and $R : (q_y, d_y)$ respectively, where q_x, q_y are conditions and d_x, d_y are scores (Definition 4), *syntactic preference overriding* can be defined based on the structure of q_x, q_y [81, 92]. We can view a preference P_x as a possible conjunctive query, which selects tuples from R that satisfy q_x. Intuitively, a tuple preference defines a *class* of entities with some particular features. If P_x explicitly refers to a *subclass* of the entities that P_y refers to, whenever they both apply, the more specific one, i.e. the one defining the subclass, *overrides* the more generic one. Building on the idea of conjunctive query containment and containment mappings [13, 29], preference overriding can be defined as follows [81]:

Definition 2.11. Given two preferences $R : (q_x, d_x)$ and $R : (q_y, d_y)$, P_y *is overridden by* P_x, if each atomic condition in q_y is mapped to an atomic condition in q_x with the same relations and attributes.

Then, preference P_y is called *generic* and preference P_x is called *specific*. [92] additionally requires that preference P_x is more related to the current query than P_y.

Example 14: Addison's preference for comedies can be expressed as ($movie.genre$ = '$comedy$', 0.9). In addition, Addison does not like comedies directed by Ben Stiller expressed as $movie$: ($movie.genre$ = '$comedy$' and $movie.director$ = '$BenStiller$', -0.9). The latter preference is more specific than the first one as it refers to a subclass of the comedies. Hence, when both preferences apply, the more specific preference overrides the preference for comedies.

Pareto Preference Composition. In pareto composition, preferences are considered equally important.

Definition 2.12. Let P_x and P_y be two preference relations defined over the same relational schema R. The pareto preference composition relation $\succ_{P_x \otimes P_y}$ is defined over R, such that, $\forall$ t_i, t_j of R, $t_i \succ_{P_x \otimes P_y} t_j$, if and only if, $(t_i \succ_{P_x} t_j \wedge \neg(t_j \succ_{P_y} t_i)) \vee (t_i \succ_{P_y} t_j \wedge \neg(t_j \succ_{P_x} t_i))$.

Note that for two tuples t_1 and t_2 and a preference relation P, $\neg(t_1 \succ_P t_2) \equiv (t_2 \succ_P t_1 \vee t_1 \sim_P t_2)$. Intuitively, under pareto composition, a tuple is better than (or dominates) another if it is at least as good (i.e. not worse) under one preference and strictly better under the other. For instance, given the preference graphs of two preferences P_x, P_y (Figure 2.9a, 2.9b), Figure 2.9d depicts the preference graph of $P_x \otimes P_y$. As another example, consider the following.

Example 15: Assume the preferences P_8 and P_9 defined above. The pareto preference $P_8 \otimes P_9$ can be defined as: $t_i \succ_{P_8 \otimes P_9} t_j$, if and only if, ($t_i[genre]$ = '$drama$' $\wedge$ $t_j[genre]$ = '$horror$' $\wedge$ $t_i[duration] \geq t_j[duration]$) $\vee$ ($t_i[duration] > t_j[duration] \wedge t_j[genre] \neq$ '$drama$') $\vee$ ($t_i[duration] > t_j[duration] \wedge t_j[genre]$ = '$drama$' $\wedge$ $t_i[genre] \neq$ '$horror$').

Now, for the movie relation of Figure 2.2a, under $P_8 \otimes P_9$, t_3 is preferred over t_1 and t_1, t_2 are incomparable.

Pareto composition is also applicable to relations defined over different schemas. This is called multidimensional pareto composition.

Definition 2.13. Let P_x and P_y be two preference relations defined over the relational schemas R and R' with attribute domains $dom(A)$ and $dom(A')$ respectively. The multidimensional pareto preference relation $\succ_{P_x \otimes P_y}$ defined over the Cartesian product $R \times R'$ is a subset of $dom(A) \times dom(A')$, such that $(t_i, t'_i) \succ_{P_x \otimes P_y} (t_j, t'_j)$, if and only if, $(t_i \succ_{P_x} t_j \wedge \neg(t'_j \succ_{P_y} t'_i)) \vee (t'_i \succ_{P_y} t'_j \wedge \neg(t_j \succ_{P_x} t_i))$, where t_i, t_j are tuples of R and t'_i, t'_j tuples of R'.

Pareto composition does not preserve the weak, total or strict partial orders [32]. Pareto composition is also easily extended for the "at least as preferable" relation $\succsim_{PR}$ [51]:
(i) $(t_i, t'_i) \succ_{P_x \otimes P_y} (t_j, t'_j)$, if and only if, $(t_i \succ_{P_x} t_j \wedge t'_i \succsim_{P_y} t'_j) \vee (t'_i \succ_{P_y} t'_j \wedge t_i \succsim_{P_x} t_j)$, (ii) $(t_i, t'_i) \simeq_{P_x \otimes P_y} (t_j, t'_j)$, if and only if, $(t_i \simeq_{P_x} t_j \wedge t'_i \simeq_{P_y} t'_j)$ and (iii) $(t_i, t'_i) \,||_{P_x \otimes P_y}$ (t_j, t'_j) otherwise.

Pair-wise Comparisons Preference Composition. In the context of rankings aggregation, in 1785, [36] outlines a generic method designed to simulate pair-wise elections in a voting system. Given two elements, or objects, the first one precedes the second one if it precedes it in the majority of rankings.

In a similar way, we can say that, given a set of preferences, a tuple t_i is preferred over a tuple t_j, if and only if, t_i is preferred over t_j for the majority of the preferences. Formally:

Definition 2.14. Let S_P be a set of preference relations defined over the same relational schema R. S_P is divided into two sets S_{P_1} and S_{P_2}, such that $S_{P_1} \bigcap S_{P_2} = \emptyset$ and $S_{P_1} \bigcup S_{P_2} = S_P$. Given two tuples t_i, t_j of R, such that $t_i \succ_{P_x} t_j$, $\forall P_x \in S_{P_1}$, and $\neg(t_i \succ_{P_y} t_j)$, $\forall P_y \in S_{P_2}$, the pair-wise comparisons preference composition relation $\succ_{pw}$ is defined over R, such that $t_i \succ_{pw} t_j$, if and only if, $|S_{P_1}| > |S_{P_2}|$.

Other similar methods can also be found in voting theory. For example, given a set of rankings, a very simple approach considers only the first tuple of the individual rankings to construct the aggregate one; tuples are ordered according to the number of times that each one appears in the first position of the rankings. This solution seems to be not fair, since it ignores all the non first positions of a tuple. To overcome this problem, [19] introduces a method in which the position of a tuple in the aggregate ranking is determined by the sum of its positions in the initial rankings.

Set-Oriented Preference Composition. The following ways of combining preferences are only applicable to preferences defined over the same relational schema. They

correspond to the intersection, set difference and union of preference relations. Recall that preference relations are defined as binary relations, i.e. as sets.

Given the relations P_x and P_y, the intersection preference relation $\succ_{P_x \blacklozenge P_y}$ corresponds to $\succ_{P_x} \cap \succ_{P_y}$. More precisely:

Definition 2.15. Let P_x and P_y be two preference relations defined over the same relational schema R. The intersection preference relation $\succ_{P_x \blacklozenge P_y}$ is defined over R, such that, $\forall\ t_i, t_j$ of R, $t_i \succ_{P_x \blacklozenge P_y} t_j$, if and only if, $t_i \succ_{P_x} t_j \wedge t_i \succ_{P_y} t_j$.

Example 16: The intersection preference $P_8 \blacklozenge P_9$ can be expressed as: $t_i \succ_{P_8 \blacklozenge P_9} t_j$, if and only if, $(t_i[genre] = \text{`}drama\text{'} \wedge t_j[genre] = \text{`}horror\text{'}) \wedge (t_i[duration] > t_j[duration])$. Therefore, under $P_8 \blacklozenge P_9$, t_3 is preferred over t_2.

The difference preference relation $\succ_{P_x - P_y}$ corresponds to set difference $\succ_{P_x} - \succ_{P_y}$, while the union preference relation $\succ_{P_x + P_y}$ corresponds to union $\succ_{P_x} \cup \succ_{P_y}$. Thus, formally:

Definition 2.16. Let P_x and P_y be two preference relations defined over the same relational schema R. The difference preference relation $\succ_{P_x - P_y}$ is defined over R, such that, $\forall\ t_i, t_j$ of R, $t_i \succ_{P_x - P_y} t_j$, if and only if, $t_i \succ_{P_x} t_j \wedge \neg(t_i \succ_{P_y} t_j)$.

Definition 2.17. Let P_x and P_y be two preference relations defined over the same relational schema R. The union preference relation $\succ_{P_x + P_y}$ is defined over R, such that, $\forall\ t_i$, t_j of R, $t_i \succ_{P_x + P_y} t_j$, if and only if, $t_i \succ_{P_x} t_j \vee t_i \succ_{P_y} t_j$.

Strict partial order is preserved by intersection but not by set difference or union. None of the set-oriented composition operators preserve either the weak or the total order [32].

Transitive Closure. The transitive closure of a preference relation is defined as follows [32].

Definition 2.18. Let $\succ_{PR}$ be a preference relation defined over a relational schema R and t_i, t_j be two tuples of R. The transitive closure of $\succ_{PR}$ is a preference relation $\succ_{PR^*}$ over R defined as: $t_i \succ_{PR^*} t_j$, if and only if, $t_i \succ^n_{PR} t_j$, $n > 0$, where:

(i) $t_i \succ^1_{PR} t_j \equiv t_i \succ_{PR} t_j$ and

(ii) $t_i \succ^{n+1}_{PR} t_j \equiv \exists t_k$, such that $t_i \succ_{PR} t_k \wedge t_k \succ^n_{PR} t_j$.

An important point is that, when preference relations are defined using formulas, the transitive closure is not defined as the closure of a finite relation as is the case of a database instance. [102] introduces a constraint language for expressing the preference formulas PFs that allows comparison and a limited form of arithmetic. They prove that the transitive closure computation of a partial order preference relation expressed using their language terminates. They also provide estimations for the size of the transitive closure for pareto, prioritized and intersection composition.

Quantitative Composition

Numerical preference composition can only be applied to quantitative preferences.

Definition 2.19. Given two preferences P_x, P_y over R defined through preference functions f_{P_x}, f_{P_y} respectively and a combining function $F : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$, $\forall\ t_i,\ t_j$ in R, $t_i \succ_{rank_F(P_x,P_y)} t_j$, if and only if, $F(f_{P_x}(t_i), f_{P_y}(t_i)) > F(f_{P_x}(t_j), f_{P_y}(t_j))$.

To assign importance to preferences, a weighted function can be used.

Example 18: Assume preference P_3 with scoring function $f_{P_3}(t_i) = 0.001 \times t_i[duration]$ and preference P_{10} with scoring function $f_{P_{10}}(t_i) = 0.0001 \times t_i[year]$. The numerical preference $rank_F(P_3, P_{10})$ with combining function $F(f_{P_3}(t_i), f_{P_{10}}(t_i)) = 0.1 \times f_{P_3}(t_i) + 0.9 \times f_{P_{10}}(t_i)$ assigns weight 0.1 to P_3 and weight 0.9 to P_{10}. Under this preference, tuples t_1, t_2, t_3 get the scores 0.185, 0.187, 0.199 respectively and so, t_3 is preferred over t_2 which in turn is preferred over t_1.

Other types of combining functions, besides the weighted one, are the "*min*" and the "*max*" functions. If preferences are defined through conditions, then *conjunctive* and *disjunctive* preferences can be constructed by combining the corresponding conditions through the boolean operators *and* and *or* respectively. The interest score of a complex preference is computed as a function of the scores of the participating preferences.

[80] defines three classes of combining (or ranking) functions: (*i*) *inflationary*, where the preference in a tuple that satisfies multiple preferences together increases with the number of these preferences; (*ii*) *dominant*, where the most important preference dominates; and (*iii*) *reserved*, where the preference in a tuple is between the highest and the lowest degrees of interest among the preferences satisfied.

Analogously to prioritized and pareto composition, numerical composition may be applied to preferences defined over different schemas (e.g. [62, 67]).

Definition 2.20. Given two preferences P_x, P_y defined over R, R' through preference functions f_{P_x}, f_{P_y} respectively and a combining function $F : \mathbb{R}\times\mathbb{R} \rightarrow \mathbb{R}$, $(t_i, t'_i) \succ_{rank_F(P_x,P_y)} (t_j, t'_j)$, if and only if, $F(f_{P_x}(t_i), f_{P_y}(t'_i)) > F(f_{P_x}(t_j), f_{P_y}(t'_j))$, where t_i, t_j are tuples of R and t'_i, t'_j are tuples of R'.

2.2.2 Combining Preferences of Different Granularity

The mechanisms presented in the previous section compose preferences for tuples. In this section, we study approaches for combining preferences of different granularity.

In the multi-relational preference model of [80, 82], user preferences are expressed at the tuple and relationship level. One can compose *implicit preferences*, i.e. preferences that can be implied by the known ones. Implicit preferences can be derived from composeable preferences. Two preferences P_x and P_y are *composeable*, if and only if: (*i*) P_x is a join preference of the form $R_x : (q_x, d_x)$ connecting R_x to a relation R_y, and (*ii*) P_y is a join or selection preference on R_y, i.e. $R_y : (q_y, d_y)$.

Definition 2.21. An *implicit preference* $R : (q, d)$ is defined from the composeable preferences P_x and P_y, as follows:

- $R \equiv R_x$,
- q is the conjunction of the conditions q_x, q_y and
- d is a function of the degrees of interest of the base preferences, i.e. $d = f(d_x, d_y)$.

d is a non-increasing function, such as the product and the minimum of the base degrees of interest d_x, d_y, on the ground that it cannot exceed the degrees of interest of its supporting preferences.

Example 19: Addison likes actress Julia Roberts and expresses a selection preference: $actor : (actor.name = `Julia\ Roberts', 0.8)$. Moreover, she has preferences expressed over the joins between the relations of the database schema (Figure 2.1). In particular, she considers the actor of a movie very important and hence she has the following join preferences: $movie : (movie.mid = play.mid, 1)$ and $play : (play.aid = actor.aid, 1)$. Then, we can define an implicit preference for movies with Julia Roberts (taking the product of the degrees of interest), as follows:
$movie : (movie.mid = play.mid$ and $play.aid = actor.aid$ and $actor.name = `Julia\ Roberts', 0.8)$.

[51] defines preferences at tuple and attribute level. In a qualitative form, priorities are expressed among (i) values of specific attributes and (ii) attributes themselves. Attribute preferences can be used to express priorities among tuple preferences. This is illustrated in the following example.

Example 20: Consider the movie instance of Figure 2.2a and the preferences: (i) Hitchcock is preferred to Curtiz or Spielberg (preference P_D), (ii) horror movies are preferred to drama movies (preference P_G) and (iii) the director of a movie is as important as its genre (preference P_{DG}). To combine the preferences P_D and P_G, the proposed model takes the pareto preference composition $P_D \otimes P_G$, since, as expressed in the third preference, P_D and P_G are equally important. Therefore, with regards to $P_D \otimes P_G$, t_2 is preferred to t_1 and t_3 and t_1, t_3 are incomparable.

2.2.3 A Summary of Preference Composition

Given a set of tuple preferences, there are different composition mechanisms. In general, composition may reflect different user attitudes towards resolving preference conflicts and may order a pair of tuples in a qualitative or in a quantitative way. Table 2.4 summarizes composition mechanisms for tuple preferences based on user attitude and tuple ranking.

As we have discussed earlier in this section for preference composition, every composition mechanism defined over preference relations can be also applied to preferences defined using functions (or degrees of interest) leading to a qualitative tuple ranking. Interestingly, most approaches follow a pure qualitative or quantitative approach for preference

Table 2.4: Preference composition w.r.t. tuple ranking and attitude.

		Attitude		
		Overriding	*Dominant*	*Combinatory*
Tuple Ranking	*Qualitative*	prioritized, lexicographical	−−	pareto, multidimensional pareto, pair-wise comparisons, intersection, difference, union
	Quantitative	syntactic overriding	*max*, *min*	*avg*, *weighted average*, ...

Table 2.5: Preference composition w.r.t. granularity.

	Tuple	*Relation*	*Attribute*	*Relationship*
Tuple	[11, 119, 120, 31, 32, 60, 74, 82] [80, 81, 84, 10, 112, 113, 92, 135]	−−	[51]	[82, 80, 81]
Relation		−−	−−	−−
Attribute			[51][92]	−−
Relationship				[82, 80, 81]

representation and composition. For example, if one approach represents preferences using scoring functions then it also combines preferences using scoring functions. It would be however interesting to see the intermingle of methods for representation and composition. Only [74] enriches its qualitative model with quantitative preference representation and composition mechanisms.

Apart from mechanisms for composing tuple preferences, there are also composition mechanisms for preferences of different granularity, for example, for composing preferences for tuples with preferences for attributes. Table 2.5 summarizes composition approaches based on the combination of preferences of different granularity. We observe that most approaches deal with tuple-to-tuple preference composition, whereas there are combinations that have not been touched at all. For example, one could have a preference overriding mechanism that considers a relation preference in the lack of specific preferences for its tuples.

2.3 Preferential Query Processing

Preferential query processing methods employ preferences to provide users with customized answers by changing the order and possibly the size of results. Several methods have been proposed by the database community focusing on integrating preferences into query processing. Next, we present such methods organized into the following topics.

- *Expanding database queries.* These methods deal with preferential query processing through the process of expanding regular database queries with preferences, for

example, by adding selection conditions that express user preferences on attribute values.

- *Employing preference operators.* These methods integrate preferences into query processing by augmenting database queries with special preference algebra operators. These special operators may be implemented as separate relational operators. Alternatively, the operator translation may be achieved through the process of query re-writing.

We also discuss methods for improving the performance of preferential query processing, for instance, by performing pre-processing steps off-line to construct rankings of database tuples based on preferences with the purpose of reducing the on-line query processing time.

Finally, there are special-purpose algorithms for generating the top-k results ranked according to some (combining) preference function. Since top-k processing is an important component of preferential query processing, we also present a short taxonomy of related work.

In the rest of this section, we study and compare preferential query processing techniques that expand database queries (Section 2.3.1), employ special preference operators (Section 2.3.2), pre-compute rankings (Section 2.3.3) and perform top-k computation (Section 2.3.4).

2.3.1 Expanding Database Queries

A number of approaches use the set of preferences that is provided for a particular user to expand regular database queries and rank the query results according to these preferences. In doing so, there are three fundamental steps:

- *Preference relatedness.* Determining which preferences are related and hence, applicable to a query. Different definitions are possible based on how the relationship of a preference with a query is understood.

- *Preference filtering.* Identifying which of the related preferences should be integrated into the query. These could be the most important ones based on their preference score, the most related based on some relatedness metric or all of them.

- *Preference integration.* Integrating the selected preferences into the original query to enable preferential query answering.

Preference relatedness

From a set of preferences known for a user at query time, all of them may be considered related to the query and applied on the query results (e.g. [51]) or only a subset (e.g. [80]), determined on the basis of the relationship of a preference with the query. To understand

the different ways in which such a relationship can be defined, let us consider a preference $(\mathcal{C}, P)$ and a query $(\mathcal{C}_Q, Q)$. P is defined for the context $\mathcal{C}$, which (based on our discussion in Section 2.1.3) can be internal, external (or both), or null if the preference is context-free (in which case, we would simply write P). Similarly, the user query has two parts: an internal part, Q, formulated over the database, and an external part, $\mathcal{C}_Q$, which is described by the same set of parameters, such as the time of the query, that are used for the external part of $\mathcal{C}$. $\mathcal{C}_Q$ may be null if no external context is specified.

Definition 2.22. A preference $(\mathcal{C}, P)$ is *related* to a query $(\mathcal{C}_Q, Q)$ if: (i) the external part of the context $\mathcal{C}$ *matches* the external query context $\mathcal{C}_Q$ and the internal part of the context $\mathcal{C}$ *matches* the internal query context Q, and (ii) the preference part P is *applicable to* (a subset of) Q's results.

In what follows, we elaborate each part of the definition separately.

Context Matching. Independently of the type (internal or external) of context, *context matching* can be defined with the help of a metric measuring the distance, similarity or difference of two contexts. Therefore, although the context matching methods described in the sequel have been discussed in the literature for a single type of context (either internal or external) and that is the way we are presenting them below, they are applicable to both types.

One approach is to represent both contexts as vectors and measure their similarity. [10] considers a preference $(\mathcal{C}, P)$, where $\mathcal{C}$ is internal context, and a conjunctive query Q over a relation $R = \{A_1, \ldots, A_d\}$ (no external context is specified) and take the vector representations of $\mathcal{C}$ and Q as follows. Consider the set $\mathcal{D}$ of all distinct $< attribute, values >$ pairs appearing in a relation R with size equal to N. We refer to the i-th element of $\mathcal{D}$ by $\mathcal{D}[i]$. A vector representation of a context $\mathcal{C} = \wedge_{j \in l}(A_j = a_j)$, $l \subseteq \{1, \ldots, d\}$ and $a_j \in dom(A_j)$, is a binary vector $V_\mathcal{C}$ of size N. The i-th element of the vector corresponds to $\mathcal{D}[i]$. If $\mathcal{D}[i]$ appears among the conjunctions of $\mathcal{C}$, then $V_\mathcal{C}[i]$=1; otherwise it is 0. Analogously, the vector representation of a conjunctive query is a binary vector V_Q of size N. If $\mathcal{D}[i]$ is one of the query conjuncts, then $V_Q[i] = 1$; otherwise it is 0. The similarity between $\mathcal{C}$ and Q can be defined using their vector representations $V_\mathcal{C}$ and V_Q as follows:

$$sim(\mathcal{C}, Q) = cos(V_\mathcal{C}, V_Q) = \frac{V_\mathcal{C} \cdot V_Q}{|V_\mathcal{C}||V_Q|}.$$

If the context parameters take values from hierarchical domains (e.g. the hierarchies in Figure 3.1), then we can compare contexts expressed at different levels of abstraction. For this purpose, we can exploit the notion of cover partial order between contexts [113]. For instance, the notion of coverage allows relating a context in which the parameter *time_period* is instantiated to a specific occasion (e.g. *Christmas*) with a context in which the same parameter is expressed with a more general period (e.g. *holidays*). Using the cover partial order between contexts, we can relate the external context of a preference ($\mathcal{C}$, P) to the context $\mathcal{C}_Q$ of a query, if $\mathcal{C}$ is more general than $\mathcal{C}_Q$, that is, if the context values

specified in $\mathcal{C}$ are equal to or more general than the ones specified in $\mathcal{C}_Q$. In addition, we can measure the distance between the contexts $\mathcal{C}$ and $\mathcal{C}_Q$ in the hierarchy. [113] defines the hierarchical distance between two contexts, $\mathcal{C} = (c_1, c_2, \ldots, c_n)$ and $\mathcal{C}_Q = (c_1^Q, c_2^Q, \ldots, c_n^Q)$, both defined with the help of a set of hierarchical parameters C_1, ... C_n, as:

$$dist_H(\mathcal{C}, \mathcal{C}_Q) = \sum_{i=1}^{n} d_H(L_{x_i}, L_{y_i}^Q),$$

where L_{x_i} (resp. $L_{y_i}^Q$) is the hierarchy level of value c_i (resp. c_i^Q) of parameter C_i and $d_H(L_{x_i}, L_{y_i}^Q)$ is equal to the number of edges that connect L_{x_i} and $L_{y_i}^Q$ in C_i's hierarchy.

To locate the contextual preferences with the closest context to the query context, that is, the preferences whose context is defined in the most detailed hierarchy levels among the preferences whose context is more general from the context of the query, an algorithm that is built upon a data structure, called *profile tree*, is employed. The profile tree offers a space-efficient representation of contexts by taking advantage of the co-occurrence of context values in preferences. Details about the hierarchical distance between two contexts, the profile tree and the algorithm for locating the relevant to a query preferences are given in Chapter 3.

A similar metric, which computes the *relevance index* for a preference $CP = (\mathcal{C}, P)$, is used in [92]:

$$relevance(CP) = \frac{dist(\mathcal{C}_Q, \mathcal{C}_{root}) - dist(\mathcal{C}, \mathcal{C}_Q)}{dist(\mathcal{C}_Q, \mathcal{C}_{root})},$$

where $dist(C_Q, C_{root})$ represents the highest possible distance of the query context with regards to any other context for which a preference exists. Preferences whose context is the query context, have the maximum relevance index, that is 1, while preferences whose context is the most general one based on the available hierarchies have the minimum relevance, that is 0.

Generalizing the discussion above, we can relate a context $\mathcal{C}$ to a context $\mathcal{C}'$ (which could be the external or internal query context), if $\mathcal{C}$ is relaxing zero or more parameters of $\mathcal{C}'$ in any of the following ways: a context parameter may be relaxed *upwards* by replacing its value by a more general one, *downwards* by replacing its value by a set of more specific ones or *sideways* by replacing its value by sibling values in the hierarchy. Given all these possible relaxations, appropriate distance metrics that exploit the number of relaxed parameters and the associated depth of such relaxations can be employed to measure how well the context $\mathcal{C}$ matches the context $\mathcal{C}'$ [114].

Given a function for context matching, it is possible to rank preferences based on their *relatedness score* (computed with the help of this function), which captures how well or closely a preference context matches a query context.

Preference Applicability. Context matching helps distinguish between preferences that are valid in the context of a query and preferences that are out of context. It does not guarantee that a preference can be combined with the query and yield an interesting,

non-empty output. This is a largely ignored issue by existing work in preference modeling and applications. We consider the following cases of *preference applicability*:

Instance applicability. In general, a preference P is instantly applicable to a query Q if Q, combined conjunctively with P, is executed over the current database instance and its result set is not empty. For example, consider a query about recent movies and a preference for movies directed by Steven Spielberg. This preference is instantly applicable only if the database contains the recent entries of the specified director.

Semantic applicability. To decide whether a preference is semantically applicable to a query, additional knowledge, outside the database, is needed. Consider as an example a query about comedies. Then, a preference for movies directed by Woody Allen is applicable since this director has directed comedies. On the other hand, a preference for Andrei Tarkovsky is not semantically applicable to the same query, and, if conjunctively combined with it, no results will be returned. It is noteworthy that when a preference P is instantly applicable to a query Q, then P is also semantically applicable to Q. The reverse does not apply. As an example, consider a query about recent movies (Q) and a preference for movies directed by Quentin Tarantino (P). P is semantically applicable to Q, because we could have the knowledge that Quentin Tarantino has recently directed a movie, but, assuming that our database is not immediately updated, P is not instantly applicable to Q.

Syntactic applicability. The decision about when a preference P is syntactically applicable to a query Q is determined with respect to their structure, i.e., the relations, attributes and attribute values P and Q contain. In the context-free preference model of [82], a preference P for the tuples of a relation R is applicable (and thus related) to a query Q, if R is referenced in Q, and P is expressed over a multi-value attribute in Q. For example, if a query returns movies starring Julia Roberts, then a preference for the actor Ben Stiller is syntactically applicable, since a movie has many actors. On the other hand, assume a query about movies after 2000. Then, a preference for movies before 1990 is conflicting. This is a looser definition than the one above, since the set of semantically applicable preferences for a query is a superset of the syntactically applicable ones. Its advantage is that it enables faster identification of related preferences by comparing their structure with the structure of the query instead of possibly enumerating and comparing result sets.

Example 21: Consider the *time_period* context hierarchy of Fig. **??**. Addison wants to see an English speaking horror movie at Christmas. This could be a query like:
$(time\_period = `Christmas',$
$SELECT\ title\ FROM\ movie\ WHERE\ genre = `horror'\ AND\ language = `English').$
Addison's preferences are:
(CP_1) $(All,\ genre = `adventure')$
(CP_2) $(time\_period = `Holidays',\ language = `Greek')$
(CP_3) $(time\_period = `Holidays',\ director = `Hitchcock')$

Of these three preferences, the last two are more closely related to the query since the distance of the preference context in the *time_period* hierarchy is the smallest. It is easy to see that (CP_2) is not applicable to the query based on the syntactic characteristics of both the query and the preference: the latter is for Greek speaking movies, while the former for English speaking ones. (CP_3) is syntactically applicable. Furthermore, it is instantly and semantically applicable, since there exist English speaking horror movies in our database.

Preference filtering

All preferences related to a query may be used for ranking and selecting the tuples returned by the query. Alternatively, we can rank preferences using either their preference score (showing their intensity as we have discussed in Section 2.1 for quantitative preferences) or their relatedness score (capturing the degree to which a preference is related to a query based on some context matching function). Subsequently, we can select the top preferences based on their intensity or their relatedness to the query for modifying the query results.

Filtering based on Preference Score. We consider approaches that deal with quantitative preferences. User preferences that are related to a given query can be ordered in decreasing preference score and the top K preferences are selected for expanding the query.

[82] proposes an algorithm for extracting the top K related selection preferences at query time from a set of preferences U (called the user profile). These preferences include preferences stored explicitly in the user profile but also implicit ones that can be derived through preference composition (Section 2.2.2). The *Preference Selection Algorithm* (Algorithm 1) takes as input a query Q, a set of preferences U and an interest criterion CI, and generates the set $\mathcal{P}_K$ of top K related preferences derived from U. Various types of interest criteria CI can be applied to determine K. For example, preferences with degrees of interest greater than a threshold value or at most x preferences should be output.

The algorithm starts from the preferences that are stored in the user profile and are related to the query and iteratively considers additional preferences that are composeable with those already known. QP is the set of preferences related to the query in order of decreasing degree of interest. The set $\mathcal{P}_K$ of selection preferences related to the query is also kept ordered. Initially, QP contains all related preferences explicitly stored in the profile. The authors use the term "not conflicting" for preferences that are syntactically applicable. At each round, the algorithm picks from QP the candidate preference P with the highest degree of interest, which is processed based on its type:

- A selection preference is added in $\mathcal{P}_K$ if it satisfies the interest criterion.
- A join preference is used to compose other preferences. The algorithm considers all stored preferences that are composeable with it to infer implicit preferences that

Preference Selection Algorithm

Input: A query Q, a set of preferences U, an interest criterion CI.

Output: A set of preferences $\mathcal{P}_K$.

```
1: Begin
2: P_K = {}, QP = {};
3: for all preferences P_i ∈ U related to Q do
4:    if (P_i is conflicting with Q) then
5:       discard P_i;
6:    else
7:       QP ← P_i;
8: while QP not empty do
9:    get head P from QP;
10:   if (P is selection) then
11:      if (CI(P_K ∪ {P}) = true) then
12:         P_K ← P;
13:      else
14:         stop;
15:   if (P is join) then
16:      if (CI(P_K ∪ {P}) = true) then
17:         for all preferences P_i ∈ U composable with P do
18:            if (CI(P_K ∪ {P ∧ P_i}) = false) then
19:               exit for;
20:            if ¬ ((P_i joins to relation R ∈ P or R ∈ Q) or (P ∧ P_i is conflicting with Q)) then
21:               QP ← P ∧ P_i;
22:      else
23:         stop;
24: End
```

Algorithm 1: Preference Selection Algorithm

can be applied to the query. These are inserted into QP unless pruning takes place. This occurs under the following circumstances: (i) the new preference expands to a relation already existent in P or to a relation belonging to the query Q, in which case a cycle is generated; (ii) it is conflicting with the query Q; (iii) it does not satisfy the interest criterion.

The algorithm stops when no other preferences that satisfy the interest criterion can be derived from the profile and returns the top-K related preferences from $\mathcal{P}_K$. The monotonicity property of the implicit preference scores guarantees that any unseen preferences cannot have higher score.

[83] views the problem of preference filtering as an optimization problem with constraints. The problem parameters are the desired degree of interest in the query tuples, the desired number of tuples to be returned and the execution cost. Preference filtering is modeled as a state space search problem and a number of algorithms are proposed that

find the optimal subset of related preferences that matches the problem constraints.

Filtering based on Preference Relatedness. As we have discussed in the previous subsection, it is possible to define a function for comparing and matching contexts, and for relating a preference to a query. For example, contexts can be matched using the cosine similarity [10]. If the context parameters take values from hierarchical domains, then we can use distance metrics that combine the number of parameters in which the contexts differ and the level at which such differences occur in the context hierarchies [114]. Given a context matching function, it is possible to rank preferences based on their relatedness, i.e. based on how well their context match the query context, and select the most related preferences, i.e. whose context is the closest or the most similar to the query context (e.g. [113, 114, 92]).

Preference Integration through Query Rewriting

Preferences expressed with the help of query conditions (e.g. [82, 51]) can be naturally integrated into a query. Query rewriting approaches leverage the power of SQL to return results that satisfy the user preferences for a given query and rank them accordingly.

Given a regular database query Q and a set of user preferences U, [82] uses the top K user preferences (in order of preference) that are related to the query to modify it (a process termed *query personalization*) and generate personalized results that satisfy at least L of the K preferences. The parameters K and L provide a quantitative way to describe the desired answer. Parameter K determines the desired extent of personalization. It also provides a way to control the cost of query personalization, as the fewer the preferences integrated into a query, the more efficient that query typically is. Parameter L captures the minimum number of user criteria (i.e. preferences) that an answer should meet, thereby offering a degree of flexibility to personalizing a query answer.

Two different query re-writing mechanisms are proposed. The first one composes a single query that defines a conjunction of the initial query qualification with the disjunction of all possible conjunctions of the L conditions from the K preferences. In the second mechanism, K queries are formulated, each one augmenting the initial query with one of the K preferences. The partial results are grouped and each tuple that appears at least L times is output. In both cases, the query execution returns a ranked list of results according to the preferences they satisfy.

[51] proposes the concept of constructing blocks, or groups, of equivalent queries, each block consisting of a set of queries that generate equally preferable results. More precisely, given a query and a set of related preferences defined among values of specific attributes and attributes themselves, a *query lattice* is constructed. Taking into account only the values specified in preferences, the query lattice has one node for each combination of values of different attributes. For example, for the preferences: (*i*) Hitchcock is preferred over Curtiz or Spielberg, (*ii*) horror movies are preferred over drama movies and (*iii*) the director of a movie is as important as its genre, the query lattice of Figure 2.10 is

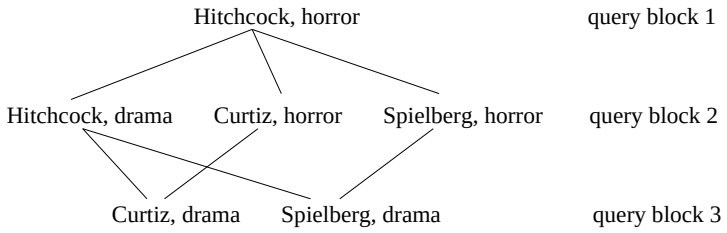


Figure 2.10: Query lattice example.

constructed. Each node in the lattice corresponds to a query. Queries are executed in an order that reflects their position in the lattice. All queries that belong to a specific query block are considered to be equivalent. To produce ranked results according to the user preferences, the queries of each group are successively executed starting from the queries of the top block and going down the lattice. Coming back to our example, and for the instance shown in Figure 3.2, t_2 is the result of the query of the first block, the queries of the second block return no results and the third block returns the tuples t_1 and t_3. Consequently, the *Lattice-based Algorithm* takes as input a relation R and a set of preferences defined over R and progressively reports a sequence of blocks of tuples of R, ensuring that each block contains tuples preferred over the tuples in the following blocks.

Summary of Expanding Queries Methods

Table 2.6 summarizes approaches that expand regular database queries with preferences using three axes: preference relatedness, filtering and integration. We observe that only a handful of approaches care about preference relatedness. Most existing approaches, whose characteristics are summarized in subsequent sections, assume all preferences known at query time as related and focus on other aspects of preferential query processing, such as the efficient execution of preference queries.

Although existing approaches on contextual preferences have focused on either internal or external context, real preferences may be defined for contexts that have both an internal and an external part. In this case, one could use a combining function that aggregates the partial relatedness scores to a single score that captures how well the internal and external parts of the preference's context match the query context. Preference applicability, i.e. when a preference can be combined with a query, is studied in [80, 81, 82] and [119]. [119] deals with query expansion based on external contextual preferences where both contexts and preferences are defined through description logic concept expressions. In this approach, contextual preferences are considered relevant to a query if their contexts are same to or more general than the query context and their preferences contain concepts which can be mapped to certain relations of the query. Then, the preference parts of such contextual preferences are used to augment the query with additional conditions.

Preference filtering can be performed either based on the preference score or based on the relatedness score, which shows how specific is a preference in a given query context. It is also possible to combine the preference and relatedness scores to compute preferences

Table 2.6: A taxonomy of approaches that use preferences for expanding database queries.

	Preference Relatedness			Preference Filtering		Preference Integration	
	All Preferences	Context Matching	Preference Applicability	Preference Score	Preference Relatedness	Top K Queries	Order All Queries
[10]		internal			✓		
[113, 114]		external			✓	✓	
[92]		external			✓	✓	
[80, 81, 82, 83]			✓	✓		✓	
[51]	✓						✓
[119]		external	✓		✓		✓

for the tuples returned by the query (and the attributes to be projected in these tuples). [92] provides examples of such combining functions, although they have not been used in practice.

2.3.2 Employing Preference Operators

Preferences can be embedded into query languages through a basic preference operator that selects from its input the set of the most preferred tuples. This operator is called winnow in [32], Best in [117] and preference selection operator in [74].

There are two fundamentals approaches to handle the preference-related operators:

- *Operator implementation.* These are special evaluation and optimization algorithmic techniques that implement the operators inside the database engine.
- *Operator translation.* Preference operators are translated into other, existing relational algebra operators.

Defining Preference Operators

The *winnow operator* is a basic preference operator for picking from an instance r the set of the most preferred tuples under a given preference relation P [32]. Formally:

Definition 2.23. Given an instance r of a relational schema R and a preference relation P over R, the winnow operator $w_P(r)$ is defined as: $w_P(r) = \{t_i \in r \mid \nexists t_j \in r$, such that $t_j \succ_P t_i\}$.

A database tuple t_i belongs to the winnow if it is not "killed" or "dominated" by another tuple t_j, that is, if no other tuple t_j in r is preferred over t_i. Clearly, winnow can

be used to select tuples for more than one relation by applying it to the result of queries defined over more than one relation.

From the definition, $win_P(r) \subseteq r$. It has been shown [32], that for every finite, nonempty instance r of R, if $\succ_P$ is a strict partial order, then $win_P(r)$ is nonempty.

For any two tuples t_i and t_j of r that belong to $win_P(r)$, it holds that $t_i \sim t_j$, that is, they are indifferent. When the preference relation $\succ_P$ is a total order, the winnow $w_P(r)$ includes just one tuple, whereas when the preference relation $\succ_P$ is a weak order, the tuples in $win_P(r)$ are the tuples that belong to the top equivalence class of r defined by $\sim$.

Partial antimonotonicity also holds when $\succ_P$ is a strict partial order: $\forall$ r_1, r_2 of R, $r_1 \subseteq r_2 \Rightarrow w_P(r_1) \supseteq w_P(r_2) \cap r_1$ [32].

A special case of winnow is called *skyline*. Generally, the skyline operator picks the tuples of a database relation that are not dominated by any other tuple in the same relation. A tuple dominates another tuple if it is as good or better with regard to a set of preferences and better in at least one preference. This is exactly the notion of pareto composition in an arbitrary number of preferences. Clearly, skylines can be expressed using winnow.

Skylines in multidimensional Euclidean spaces are first introduced in [20]. The dominance relationship is $>$ or $<$. The attributes of a relation are partitioned into `DIFF`, `MAX` and `MIN` attributes. Only tuples with identical values of all `DIFF` attributes are comparable; among those, `MAX` attribute values are maximized and `MIN` values are minimized.

Recently, several approaches redefine the typical skyline dominance definition. [27] defines the k-dominant skyline: a tuple t_i k-dominates another tuple t_j if there are k dimensions, or preferences, in which t_i is better than or equal to t_j, and t_i is better in at least one of these k dimensions. [87] proposes the k-representative skyline that selects k tuples, such that, the number of tuples that are dominated by at least one of these k tuples is maximized, while [126] introduces the ϵ-skyline that computes the set of all tuples that are not ϵ-dominated by any other tuple. Given a set of preferences, a tuple ϵ-dominates another tuple if it is as good, better or slightly worse (up to ϵ) with regard to all preferences and better in at least one preference.

Both winnow and skyline operators select the most preferred tuples from a given input set. Ranking the whole input can be achieved through the process of multiple applications of such operators. We present formally the iterated winnow operator as defined in [32].

Definition 2.24. Given an instance r of a relational schema R and a preference relation P over R, the iterated winnow operator, $win^i_P(r)$, of level i, $i > 0$, is defined as follows:

- $win1_P(r) = w_P(r)$
- $win^{i+1}_P(r) = w_P(r \text{ - } \cup^i_{k=1} win^k_P(r))$

Tuples retrieved earlier are of higher interest to the users. All tuples in any $win^i_P(r)$ are indiferrent to each other. The concept of iterated winnow operator, called Best operator, is independently defined in [118].

Implementing Preference Operators

The winnow operator can be handled by providing implementations within the query engine. A number of interesting research problems regarding the semantic optimization of relational queries that include winnow operators are introduced. For semantic optimization, a set of algebraic rules that characterize the interaction of winnow with the standard relational operators, such as commutativity, are provided in [32]. These rules can be used to optimize a query, for example by pushing selections and projections down the query tree. Further optimizations in the presence of integrity constraints, such as eliminating redundant applications of winnow, can be found in [33]. Such properties are achieved by relativizing the properties of the given preference relations to the sets of instances that satisfy the integrity constraints of the particular database schema.

Block Nested Loop (BNL)

Input: A database instance r, a preference P.
Output: $w_P(r)$.
Variables: a window W, a temporary table T.

```
1:  Begin
2:  r' = r;
3:  while r' ≠ ∅ do
4:     for every tuple t_i ∈ r' do
5:        if (∃t_j ∈ W, such that, t_j ≻_P t_i) then
6:           ignore t_i;
7:        else if (∀t_j ∈ W, such that, t_i ≻_P t_j) then
8:           remove all t_j from W;
9:           insert t_i into W;
10:       else if (t_i is indifferent to all tuples in W) then
11:          if (there is room in W) then
12:             insert t_i into W;
13:          else
14:             insert t_i into T;
15:    return the tuples from W that were inserted in W when T was empty;
16:    r' = T;
17:    empty T;
18: End
```

Algorithm 2: Block Nested Loop (BNL)

The naive way to compute the winnow of an instance r is to apply a basic nested-loop (NL) method that compares each tuple in r with every other tuple. The NL method works for every type of preference relation $\succ_P$ but requires scanning the whole r for each tuple.

A more efficient implementation is provided by the block-nested-loop (BNL) algorithm (Algorithm 2) proposed in [20] in the context of skyline queries. BNL maintains a window W of indifferent tuples. These correspond to the best tuples found so far. Since the size

Winnow for Weak Orders (WWO)

Input: A database instance r, a preference P.

Output: $w_P(r)$.

```
1: Begin
2: top = the first tuple of r;
3: win_P(r) = {top};
4: for every subsequent tuple t ∈ r do
5:    if (top ≻_P t) then
6:       ignore t;
7:    else if (t ≻_P top) then
8:       top = t;
9:       w_P(r) = {t};
10:   else if (t and top are indifferent) then
11:      w_P(r) = w_P(r) ∪ {t};
12: return w_P(r);
13: End
```

Algorithm 3: Winnow for Weak Orders (WWO)

of the window may not be enough for keeping all such tuples, the algorithm also uses a temporary table T to store any overflow tuples. Initially, the input is the instance r and W and T are empty. At each iteration, all tuples in the input are read. When a tuple t is read from the input, t is compared with all tuples in W. Three cases can occur: (i) t is dominated by a tuple in W, in which case t is discarded, (ii) t dominates one or more of the tuples in W, in which case these tuples are discarded and t is inserted into W, or (iii) t is indifferent with all tuples in W, in which case if there is room in W, t is inserted into W, otherwise t is stored in the temporary table T to be processed further in the next iteration. At the end of each iteration, all tuples added to W when T was empty are output. The next iteration uses the temporary table T as input.

BNL works correctly only when the preference relation $\succ_P$ is at least a strict partial order, since the algorithm uses transitivity. To see this, say that an input tuple t_j is dominated by a tuple t_i in W, thus t_j is discarded (case (a)). Next, a tuple t_k arrives that is dominated by t_j, but not by t_i in W. The algorithm may output t_k incorrectly.

When $\succ_P$ is a weak order, the winnow for weak orders (WWO) algorithm (Algorithm 3) [33] takes advantage of the fact that all tuples in the winnow belong to a single equivalence class. Thus, an input tuple t: (i) is dominated by all tuples in W, in which case t is discarded, (ii) dominates all of them, in which case the whole W is discarded and replaced by t, or (iii) is indifferent to all of them, in which case it is added to W. In all cases, a single comparison of t with just one tuple in W (tuple *top*) suffices. At the end of the first iteration, W will contain only tuples in the winnow.

The Sort-Filter-Skyline (SFS) algorithm (Algorithm 4) proposed in [34] for computing skylines, adds a preprocessing step to BNL that sorts all tuples in r, so that, if a tuple $t_i \succ_P t_j$, then t_i precedes t_j in the produced order. This correspond to the order produced

Sort-Filter-Skyline (SFS)

Input: A database instance r, a preference P.

Output: $w_P(r)$.

Variables: a window W, a temporary table T.

```
1:  Begin
2:  Topologically sort r based on P;
3:  r' = r;
4:  while r' ≠ ∅ do
5:     for every tuple t_i ∈ r' do
6:        if (∃t_j ∈ W, such that, t_j ≻_P t_i) then
7:           ignore t_i;
8:        else if (t_i is indifferent to all tuples in W) then
9:           if (there is room in W) then
10:             insert t_i into W;
11:          else
12:             insert t_i into T;
13:    return the tuples from W;
14:    empty W;
15:    r' = T;
16:    empty T;
17: End
```

Algorithm 4: Sort-Filter-Skyline (SFS)

by a topological sort of the preference graph of r. By processing the tuples following this order, it is ensured that when a tuple is inserted into the window W, it belongs to the winnow, thus it can be output immediately. For SFS to work, $\succ_P$ must be at least a strict partial order.

Several other approaches have been proposed for computing skylines. For example, [115] introduces the first progressive algorithm that returns skylines without scanning the whole dataset. Later, [79] proposes another progressive algorithm based on the nearest neighbor search method, while [98] introduces a branch-and-bound algorithm where datasets are indexed by an R-tree. [130, 100, 116] work on skyline computation in subspaces. The detailed presentation of these approaches is beyond the scope of this survey.

Concerning now the iterated winnow operator, a straightforward implementation of it can be achieved by applying one of the algorithms (i.e. the NL, BNL, WOW or SFS) previously described, multiple times. The first application would be on instance r to produce $win1_P(r)$ and the subsequent applications on $(r$ - $\cup_{k=1}^{i} win_P^k(r)))$, to produce $win_P^{i+1}(r)$.

A more efficient implementation of winnow and ranking is proposed in [117, 118]. The evaluating best operator algorithm (Algorithm 5) is a variation of BNL, where the repetition for computing $win_P^{i+1}(r)$ does not start from scratch each time, but instead, from those tuples that were found to be directly dominated by a tuple in $win_P^i(r)$. In

Algorithm 5, we use the notation $\beta^i_{\succ}(r)$ to refer to $win^i_P(r)$, and $D^t_{\succ}$ for the set of tuples dominated by t. The algorithm builds a special data structure, called β-tree, over the tuples of r.

The iterated winnow operator can also be implemented by topologically sorting the preference graph of r (e.g. [41, 51]).

Translating Preference Operators

Apart from the preference operators that are implemented through evaluation and optimization algorithms, there are operators that can be expressed using other relational algebra operators.

In this context, [74] defines preference queries with regards to two new relational operators: (i) the *preference selection operator* and (ii) the *grouped preference selection operator*. The preference selection operator, denoted $\sigma[P](r)$, corresponds to the winnow operator $w_P(r)$ [32]. The grouped preference selection operator applies preference selection within groups. Given a subset B of the attribute set of R, tuples in r are partitioned into groups of tuples having the same values in the grouping attributes in B. The grouped preference selection operator $\sigma[P \ group \ by \ B](r)$ selects the dominating tuples in each group. Formally:

Definition 2.25. Given an instance r of a relational schema R and a preference relation P over R, the grouped preference selection operator $\sigma[P \ group \ by \ B](r)$ is defined as: $\sigma[P \ group \ by \ B](r) = \{t_i \in r \mid \nexists t_j \in r$, such that, $t_j \succ_P t_i \wedge t_i[B] = t_j[B]\}$, where B is a subset of the attribute set of R.

Preference queries expressed using such operators are translated into standard SQL queries. [77] describes an implementation of the framework introduced in [74], using a language called Preference SQL. In particular, Preference SQL is an extension of SQL that covers all base preference constructors of [74]. Preferences are syntactically expressed inside an SQL query using the keyword PREFERRING. For example, the query:

```
SELECT * FROM movies
PREFERRING duration BETWEEN [170, 200];
```

returns movies with duration inside the interval [170, 200]. If such movies do not exist, the movies with duration closer to the interval limits are considered better. Concerning the movie instance of Figure 2.2a, the movie t_3 is the most preferable one. To execute such queries, an optimizer on top of SQL that translates Preference SQL into standard SQL is implemented. To improve the response time of preference queries, optimization techniques have been proposed in [55]. The preferences defined in [74] can also be evaluated on XML databases using the query language Preference XPATH [76].

Evaluating the Best Operator

Input: A database instance r, a preference P.

Output: A β-tree.

```
1:  Begin
2:  ctr = 1;
3:  while all tuples in r are returned do
4:     if (ctr == 1) then
5:        r' = r;
6:     else
7:        for every t_i ∈ β_≻^{ctr-1} do
8:           r' = ∪_i D_≻^{t_i};
9:     while r' ≠ ∅ do
10:       top = the first tuple of r';
11:       for every tuple t ∈ r' do
12:          if (top is indifferent to t) then
13:             insert t into a set U of unresolved tuples;
14:          else if (top ≻_P t) then
15:             insert t into a set D_≻^{top} which contains the tuples dominated by top based on P;
16:             if (t belongs to another set D_≻^{t'}) then
17:                remove t from D_≻^{t'};
18:          else if (t ≻_P top) then
19:             insert top into D_≻^{t} which contains the tuples dominated by t based on P;
20:             if (top belongs to another D_≻^{t'}) then
21:                remove top from D_≻^{t'};
22:             top = t;
23:       for every tuple t ∈ U that is not compared yet with top do
24:          if top ≻_P t then
25:             insert t into D_≻^{top};
26:             remove t from U;
27:       insert top into β_≻^{ctr};
28:       r' = U;
29:       empty U;
30:    return β_≻^{ctr};
31:    ctr++;
32: End
```

Algorithm 5: Evaluating the Best Operator

Table 2.7: A taxonomy of approaches employing preference operators.

		Implementation Level	
		Evaluation Techniques	*Operator Translation*
Query Model	*Best Answers*	winnow, skyline [32, 20, 27, 87] [33, 126, 34, 115, 79, 98, 100, 116]	preference selection, grouped preference selection [74, 77]
	Ranking	iterated winnow [32, 117, 118, 41, 51]	−−

Summary of Employing Preference Operators

To synopsize, Table 2.7 presents a taxonomy of approaches employing preference operators with regards to: (*i*) the query model, specifying if the answer includes only the best results or the whole ranked result set and (*ii*) the way that operators are implemented, specifying if evaluation techniques are employed or operators are translated into others.

Although many of the existing approaches have focused on a query model that reports the best answers, these approaches can be naturally extended in order to rank the whole result set by applying multiple times their selected processing method. As a final note, consider that the winnow operator can be handled by re-writing each preference query, i.e. a relational query containing at least one winnow operator, to exclude the appearances of winnow. This is possible, since it has been shown that winnow can be expressed using standard relational algebra operators [32]. It was also shown that winnow can be used to simulate set difference.

2.3.3 Pre-computing Rankings

Many times, the time complexity for computing personalized query answers is unacceptable for query-time operations on large databases. Motivated by this fact, recently, there are approaches proposing some pre-processing performed offline with the purpose of making online processing of queries fast. In a nutshell, such approaches employ user preferences to construct offline representative rankings of database tuples and then, at query time, select the relevant to the query rankings and use them to report results. The main focus of this section is on the processing steps performed offline, since the online phase is usually straightforward.

We organize existing approaches to answering preference queries based on materialized rankings according to the particular kind of preferences they use into:

- *Context-based* approaches, where preferences hold under specific conditions (e.g. [10, 110]).
- *Context-free* approaches, where preferences hold unconditionally, i.e. under all circumstances (e.g. [64]).

Greedy Algorithm

Input: A set of m rankings $T_m = \{\tau_1, \ldots, \tau_m\}$.

Output: A set of l representative rankings T_l.

1: **Begin**
2: $Representatives = T_m$;
3: **for** ($i = m$; $i > l$; i- -) **do**
4: $\rho = argmin_{i \in Representatives} \sum_{\tau \in T_m} d(\tau, Representatives - \{\tau_i\}) - \sum_{\tau \in T_m} d(\tau, Representatives)$;
5: $Representatives = Representatives - \{\rho\}$;
6: $T_l \leftarrow Representatives$;
7: **End**

Algorithm 6: Greedy Algorithm

Context-based Approaches

[10] and [110] propose pre-computing representative rankings of database tuples based on contextual preferences that follow either the qualitative [10] or the quantitative [110] model. [10] considers internal contextual preferences, while [110] considers external contextual preferences. Next, we present how the representative rankings are constructed in both approaches.

Given a set of contextual preferences, [10] produces a set of rankings. Initially, a ranking for each set of preferences with the same context is constructed. Instead of maintaining rankings for all different contexts, since the number of contexts that appear in preferences can be large, a method for selecting a small subset of representative rankings is proposed. To find representative rankings several algorithms can be applied. These algorithms exploit ideas such as: (i) remove at each step a ranking that is the most similar to the remaining ones (Algorithm 6) or (ii) begin with an arbitrary ranking and at each step pick the ranking which is furthest from the already selected ones (Algorithm 7).

More specifically, given the set of all rankings T_m, the *Greedy Algorithm* for finding representative rankings (Algorithm 6) removes in each iteration the ranking which when removed, causes the least increase in the total cost, where the cost for a set of rankings T_l, $T_l \subseteq T_m$, is equal to $\sum_{\tau \in T_m} d(\tau, T_l)$. The distance between a single ranking τ and a set of rankings T_l is defined as $d(\tau, T_l) = min_{\rho \in T_l} d(\tau, \rho)$, while the distance between two rankings may be computed using either the *Spearman footrule* or the *Kendall tau* [73] distance.

In a top-down approach, the *Furthest Algorithm* (Algorithm 7) starts by selecting randomly a ranking and at each step picks from the unselected rankings the one which is furthest from the already selected rankings. The algorithm continues up to collect the desirable number of representative rankings, while the remaining rankings are assigned to their closest representative.

[128] adopts the contextual preference model proposed in [10] and, based on a machine learning approach, induce a contextual total ranking that has as training data set a partial

Furthest Algorithm

Input: A set of m rankings T_m.

Output: A set of l representative rankings T_l.

```
1: Begin
2: T_l = ∅;
3: Select randomly a ranking τ ∈ T_m;
4: T_l = T_l ∪ {τ};
5: T_m = T_m − {τ};
6: for (i = 2; i < l + 1; i++) do
7:     τ = argmac_{τ'∈T_m} d(τ', T_l);
8:     T_l = T_l ∪ {τ};
9:     T_m = T_m − {τ};
10: for every τ ∈ T_m do
11:     Assign τ to its closest ranking in T_l;
12: End
```

Algorithm 7: Furthest Algorithm

ranking in a quantitative form.

A different approach for pre-computing representative rankings is introduced in [110]. The details of this work are given in Chapter 4. In a nutshell, [110], instead of creating a ranking for each different context and then maintaining a number of them, creates groups of similar preferences and produces a ranking for each group. Two different ways of defining similarity among contextual preferences are proposed. The first one considers as similar the preferences that have either the same or similar contexts, while the second one groups preferences that result in similar scores for all database tuples, i.e. preferences that have similar predicates and scores.

In both [10] and [110], when a user submits a query, the query will be matched against the representative contexts. Taking into account the similar to the query representative rankings, the query results are computed.

Context-free Approaches

In a context-free scenario, there are approaches for computing the result of a preference query based on a set of materialized rankings maintained independently of specific conditions or circumstances. Usually, such approaches employ materialized preference views, that is, relational views ordered according to a preference, or scoring, function, to compute preferential query results.

The PREFER system provides ranked answers to preference queries based on a number of pre-computed and materialized views [62, 64]. Given a relational schema $R(A_1, \ldots, A_d)$, a view v ranks the tuples of R with regard to a scoring function F_v defined as a weighted sum using a vector of weights $(w_1, \ldots, w_d)$. During the online phase, for each query Q that is also expressed via a weight vector, the view that best matches Q is selected. The view selection problem is formulated as the problem of identifying the view that needs

the least number of tuples to be fetched for computing the query answer.

Having selecting the appropriate view, PREFER returns in a pipelined way, the k tuples that maximize the query preference function. In particular, the employed algorithm computes the smallest prefix of the view to find the most preferred tuple for the query. Then, it computes a second prefix to find the most preferred tuple after the previous one and so forth, until the k most preferred tuples are retrieved. The key concept of this algorithm is the computation of a watermark value which calibrates the stopping condition in each iteration of the algorithm. Such a watermark value determines how deep in the ranked materialized view we should go to locate the top tuple of a query. The first watermark for a view v is the maximum value $T^1_{v,Q}$, such that, $\forall t \in R$, $F_v(t) < T^1_{v,Q}$ $\Rightarrow F_Q(t) < F_Q(t^1_v)$, where t^1_v is the tuple in v with the highest score. Respectively, at the second iteration of the algorithm, the tuple t^2_v, which is the tuple with the next higher score, replaces the tuple t^1_v in the process of watermark computation and so on.

Another view-based technique that also maintains ordered views based on preference functions is proposed in [37]. This work utilizes the given set of views to produce query answers, using a linear programming algorithm. [127] focuses on the reduction of the maintenance cost of the materialized top-k views, considering occurrences of deletions and updates.

Summary of Pre-Computing Rankings

Table 2.8 presents a taxonomy of the approaches that deal with preferential query processing through pre-computing and materializing rankings. We categorize such approaches with regards to two main axes: (i) preference formulation, i.e. qualitative or quantitative, and (ii) context, i.e. context-based or context-free.

The proposed context-based approaches use either internal qualitative contextual preferences [10] or external quantitative contextual preferences [110]. The main focus of both approaches is on constructing a set of representative rankings of tuples. However, these approaches are built upon a different perspective. While [10] creates initially a ranking for each different context and then, maintain some of them (the most representative ones), [110] clusters first preferences according to their similarity and then, produce a ranking for each cluster. Context-free approaches use materialized views to maintain the pre-computed rankings (e.g. [64, 37]). Most approaches that fall into this category aim at locating the k results that maximize (or minimize) a combining preference function in a pipelined manner.

2.3.4 Top-k Query Processing

Typically, a top-k query aims at providing only the top k most important results to the users. A common way to identify the k most important results is scoring all tuples based on a scoring function, possibly defined as an aggregation of a set of functions over different attributes, and reporting the k tuples with the highest scores. Although there is

Table 2.8: A taxonomy of pre-computing rankings approaches w.r.t. preference formulation and context.

		Context	
		Context-based	*Context-free*
Formulation	*Qualitative*	[10]	−−
	Quantitative	[110, 128]	[62, 64, 37, 127]

a large amount of research that addresses top-k processing techniques, we consider that the details of such techniques are out of the scope of this survey, since in most cases, they are not directly related to user preferences. Thus, below, we provide only an overview of the main approaches.A survey of top-k processing techniques in relational databases is presented in [66].

In general, methods for compounding a set of rankings to form an aggregate one, consider that each tuple in each ranking is associated with an interest score that determines its position within the ranking. Then, to construct a total ranking, instead of following the naive approach of computing the aggregate score of each tuple and ranking the tuples based on these scores, several more efficient algorithms have been proposed.

A fundamental algorithm, called FA algorithm, for retrieving the top-k tuples of a relational schema R is proposed in [49]. This algorithm considers two types of available tuple accesses: the *sorted* access and the *random* access. Sorted access enables tuple retrieval in a descending order of their scores, while random access enables retrieving the score of a specific tuple in one access. Next, we present the main steps of the FA algorithm.

- First, do sorted access to each ranking until there is a set of k tuples, such that each of these tuples has been seen in each of the rankings.
- Then, for each tuple that has been seen, do random accesses to retrieve the missing scores.
- Compute the aggregate score of each tuple that has been seen.
- Finally, rank the tuples based on their aggregate scores and select the top-k ones.

FA is correct when the aggregate tuple scores are obtained by combining their individual scores using a monotone function. This is also holds for the TA algorithm [49]. TA ensures further that its stopping condition always occurs at least as early as the stopping condition of FA. The main steps of the TA algorithm are the following:

- First, do sorted access to each ranking. For each tuple seen, do random accesses to the other rankings to retrieve the missing tuple scores.

Table 2.9: A taxonomy of top-k query processing techniques.

		Implementation Level	
		Application Level	*Within Engine*
Query Model	*Top-k Tuples*	[48, 49, 97, 52, 53]	−−
	Top-k Join Tuples	[96]	[65, 67]
	Top-k Groups of Tuples	−−	[85]

- Then, compute the aggregate score of each tuple that has been seen. Rank the tuples based on their aggregate scores and select the top-k ones.
- Stop to do sorted accesses when the aggregate scores of the k tuples are at least equal to a threshold value that is defined as the aggregate score of the scores of the last tuples seen in each ranking.

[97] and [52] independently propose algorithms equivalent to the TA algorithm. Several TA modifications with regards to the access type that can be applied have been introduced (e.g. [49, 53]). For example, the NRA algorithm is appropriate when random accesses are expensive or impossible and so, only sorted accesses are employed, while the CA algorithm is appropriate when random accesses are expensive relative to sorted accesses [49].

All the above approaches focus on constructing an aggregate ranking by combining a set of rankings that contain the same set of tuples. Clearly, in this case, the produced ranking consists of the same tuple set. Apart from such approaches, there are algorithms (e.g. [96, 65, 67]) for aggregating rankings that contain different sets of tuples. In this category of algorithms, tuples of different rankings are joined together with respect to specific join conditions. The produced ranking here, consists of a set of joined tuples, each one with an aggregate score computed from the scores of the participating tuples. Instead of working on individual tuples, [85] proposes a method that reports the k groups of tuples with the largest interest scores, where scores are computed using a group aggregation function (e.g. sum).

In Table 2.9, we taxonomy top-k query processing techniques with respect to two main axes: (i) the *query model* and (ii) the *implementation level*. Query model defines the kind of results returned by the top-k computation, that is, single tuples (e.g. [48, 97]), joined tuples (e.g. [96, 65]) or groups of tuples (e.g. [85]), while the implementation level defines the level of integration with database systems, that is, at application level outside the database engine (e.g. [49, 97, 52]) or within the query engine (e.g. [65, 67]).

2.4 Preference Learning

Learning and predicting preferences in an automatic way has attracted much current attention in the areas of machine leaning, knowledge discovery and artificial intelligence.

Although the primary focus of this survey is on representation, composition and application of preferences in databases, we present a short overview of some representative approaches to preference learning, since this is an interesting and active related area of research with potential applications in data management.

Approaches to preference learning can be classified along various dimensions. Depending on *the model* learned, we may distinguish between learning pairwise orderings of items (i.e., qualitative preferences) and learning a utility function (i.e., quantitative preferences). Depending on *the type of information provided as input*, the learning algorithm may or may not use as input positive and/or negative examples, in analogy to supervised and unsupervised learning. Along the input classification dimension, we may also have learners that use or do not use feedback from the users, which is usually in the sort of relevance judgment. Yet another distinction is on whether the input data is in the form of pairwise preferences, that is, the input is an order relation, or the input data is a set of items for which some knowledge about the associated preference degree is available. Another dimension for differentiating preference learning is based on *the data mining task* they use (such as associative rule mining, clustering or classification) as well as *the type of method* used for the specific task (such as an SVM or an a-priori method). Finally, the application also determines the preference learning process, since it affects both the form of input data (for example, clickthrough data or user ratings) as well as the desired use of the learned preferences (for example, personalized search results or recommendations).

Most methods for preference learning utilize information of the past user interactions in the form of a history or log of transactions. [70] utilizes as input data logs of the form of user clickthrough data, namely the query-logs of a search engine along with the log of links that the user actually clicked on from those in the presented ranked list. The fact that a user clicked on a link l_i and did not click on a link l_j ranked higher in the list than l_i, is interpreted as a user preference of l_i over l_j. Using clikthrough data as training data, the goal is to learn a ranking function that, for each query q, produces an order of links based on their relevance to q. This is achieved by selecting the function that produces the order having the minimal distance from the orders inferred from the clickthrough data, analogous to classification by minimizing training errors. A support vector machine (SVM) algorithm is used.

User logs are also used as input in [59], but in the form of relational instances. Since there is no explicit ranking information in the log of relations, to detect preferences, the frequencies of the different attribute values, i.e., their number of entries, in the log relation are used. Then, x is preferred over y, if and only if, $freq(x) > freq(y)$. Preferences between values of individual attributes are used to infer positive and negative preferences, numerical preferences and complex preferences [74]. An important assumption, especially for learning negative preferences or dislikes, is the close word assumption indicating that a user knows all possible values of an attribute. Otherwise, the fact that a value x does not appear in the log could be interpreted as either a dislike of x or lack of knowledge of x. [68] extends this work to address the problem of contextual user preferences mining.

The problem of learning negative preferences from the log is addressed by a statistical approach for modeling user preferences in [71]. The main idea is to also consider the popularity of a specific item in the overall dataset, called *accessibility* of the item. The frequency with which each specific user has selected the item is called *selection probability*. Then, the preference is determined by the selection probability divided by the accessibility.

The general problem of learning a preference relation in the form of partial order given a set of examples is discussed in [125] and [69]. In the problem formulation defined in [125], there is a general partial order $\mathcal{R}$ and each user u refines it by defining an order $\mathcal{R}_u$, such that, $\mathcal{R}_u \supseteq \mathcal{R}$. The problem is: given a point p, find for which $\mathcal{R}_u$ p is a skyline. [69] studies the following problem: given two set of items, a set of superior examples and a set of inferior examples, find a strict partial order, such that, every item is dominated by at least one item in the set of superior examples and it is not dominated by any other item in the the set of inferior examples.

The approach proposed in [35] is an example of employing user feedback to improve preference learning. They consider the problem of learning how to rank items given the feedback that an item should be ranked higher than another. For a set of items $\mathcal{I}$, the employed preference function $Pref(i_1, i_2)$, $Pref$: $\mathcal{I} \times \mathcal{I} \rightarrow [0, 1]$, returns a value indicating which item, i_1 or i_2, is ranked higher. The learning phase of such a function takes place in a sequence of rounds. At each round, items are ranked with respect to $Pref$. Then, the learner receives feedback from the environment. The feedback is assumed to be an arbitrary set of rules of the form "i_1 should be preferred to i_2". Given that $Pref$ is a linear combination of n primitive functions, i.e., $Pref(i_1, i_2) = \sum_{i=j}^{n} w_j F_j(i_1, i_2)$, at each round the weights w_j are updated with regards to the $loss$ of a function F with respect to the user feedback, where $loss$ is the normalized sum of disagreements between function and feedback. The learned function $Pref$ is used to rank new items. For a new set $\mathcal{I}'$, $Pref(i_1, i_2)$ is evaluated on all pairs $i_1, i_2 \in \mathcal{I}'$. Since the result pairwise preference judgments may produce more than one rankings, the ranking that maximizes the agreements with the learned preference function is selected.

Moreover, applying machine learning techniques for learning ranking functions has recently attracted much attention in the research literature (e.g., [25, 132, 131]).

From a different perspective, recommendation systems exploit logs of the form of ratings of items provided by users in the past for providing users with items of potential interest [9]. Formally, let $\mathcal{U}$ be the set of all users and $\mathcal{I}$ be the set of all possible items to be recommended. A utility function $\mathcal{F}$, similar to the preference function (Definition 3), measures the usefulness of an item i to a user u, $\mathcal{F}$: $\mathcal{U} \times \mathcal{I} \rightarrow \mathcal{R}$, where $\mathcal{R}$ is a totally ordered set representing the rating of the user for the item. Since the utility function is usually not defined on the whole $\mathcal{U} \times \mathcal{I}$ space, the central problem is how to automatically predict the ratings for items that the user has not previously rated. There are two general approaches. In *content-based recommendations* (e.g., [99, 95]), the user will be recommended items similar to the ones they have rated high in the past. Similarity between items depends on the type of items. In general, in the case of relational instances,

one can view them as vectors and use an appropriately defined similarity function. In *collaborative recommendations* (e.g., [78, 22]), a user u will be recommended items that users similar to u have rated highly. In this case, each user is considered as a vector in the dimensions of items.

2.5 Summary

Synopsizing, the focus of this review is mainly on the representation, composition and use of preferences in databases. Although there is a large number of algorithms for the implementation of preference queries, especially for top-k and skyline queries, we aim at providing only an overview of the main approaches for different types of such queries. Finally, we refer to preference learning approaches.

A version of this chapter appears as a survey article in [108].

Chapter 3

Adding Context to Preferences

Personalized information delivery aims at addressing the explosion of the amount of data currently available to an increasingly wider spectrum of users. Instead of overwhelming the users with all available data, personalization systems provide users with only the data that is of interest to them. Preferences have been used as a means to address this challenge. To this end, a variety of preference models have been proposed most of which follow either a qualitative or a quantitative approach. For instance, using a qualitative model, users may explicitly state that they prefer visiting archaeological sites than science museums, while in a quantitative approach, a preference in archaeological sites may be expressed by assigning high scores to such places.

However, most often users have different preferences under different circumstances. For instance, the current weather conditions may influence the place one wants to visit. For example, when it rains, a museum may be preferred over an open-air archaeological site. *Context* is a general term used to capture any information that can be used to characterize the situations of an entity [39, 16]. Common types of context include the *computing context* (e.g. network connectivity, nearby resources), the *user context* (e.g. profile, location), the *physical context* (e.g. noise levels, temperature) and *time* [30, 18].

In this chapter, we propose enhancing preferences with context-related information. We use context to indicate any attribute that is not part of the database schema. Context

is modeled using a set of multidimensional context parameters. A specific context state or situation corresponds to an assignment of values to context parameters. By allowing context parameters to take values from hierarchical domains, different levels of abstraction for the captured context data are introduced. For instance, the context parameter *location* may take values from a *city*, *country* or *continent* domain. Users employ context descriptors to express their preferences on specific database instances for a variety of context states expressed with varying levels of detail.

Each query is also associated with one or more context states. The context state of a query may, for example, be the current state at the time of its submission. Furthermore, a query may be explicitly enhanced with context descriptors to allow exploratory queries about hypothetical context states. A central problem is identifying those preferences that are applicable to the context states that are most relevant to the states of a query. We call this problem *context resolution*. Context resolution is divided into two subproblems: (a) the identification of all candidate context states that encompass the query states and (b) the selection of the most appropriate states among these candidates. The first subproblem is resolved through the notion of the cover partial order between states that relates context states expressed at different levels of abstraction. For instance, the notion of coverage allows relating a context state in which location is expressed at the level of a *country* and a context state in which location is expressed at the level of a *continent*. To resolve the second subproblem, we consider appropriate distance metrics that capture similarity between context states.

We introduce algorithms for context resolution that build upon two data structures, namely the preference graph and the profile tree, that index preferences based on their associated context states. The *preference graph* explores the cover partial order of context states to organize them in some form of a lattice. A top-down traversal of the graph supports an incremental specialization of a given context state, whereas a bottom-up traversal an incremental relaxation. The *profile tree* offers a space-efficient representation of context states by taking advantage of the co-occurrence of context values in preferences. It supports exact matches of context states very efficiently through a single root-to-leaf traversal.

Our focus is on managing context for preferences, i.e. expressing, storing and indexing contextual preferences. In general, preferences may be collected using various ways. Preferences may be provided explicitly by the users or constructed automatically, for instance, based on the past behavior of the same or similar users. Such methods for the automatic construction of preferences have been the focus of much current research (e.g. [94]). A practical way to create profiles that we have used in our experiments is to assemble a number of default profiles and then ask the users to update them appropriately.

We have evaluated our approach along two perspectives: usability and performance. Our *usability experiments* consider the overhead imposed to the users for specifying context-dependent preferences versus the quality of the personalization thus achieved. We used two databases of different sizes. The sizes of the database have two important

implications for usability. First, they affect the number of preferences. Then, and most importantly, they require different methods for evaluating the quality of results. Our *performance experiments* focus on our context resolution algorithms that employ the proposed data structures to index preferences for improving response and storage overheads.

In a nutshell, in this chapter, we

- propose a model for annotating preferences with contextual information; our multi-dimensional model of context allows expressing contextual preferences at various levels of detail,
- formulate the problem of context resolution, as the problem of selecting appropriate preferences for personalizing a query based on context,
- present data structures and algorithms for implementing context resolution and
- evaluate our approach in terms of both usability and performance.

The rest of this chapter is structured as follows. In Section 3.1, we present our context and preference model, while in Section 3.2, we formulate the context resolution problem. In Section 3.3, we introduce data structures used to index contextual preferences and algorithms for context resolution and finally, Sections 3.4 and 3.5 present our usability and performance evaluation results, respectively.

3.1 Contextual Preferences

We propose annotating preferences with specifications regarding the context states under which they hold. We present first, our model for specifying context states and then, introduce contextual preferences, e.g. preferences annotated with context information.

In the rest of this chapter, we explain all our definitions with examples that refer to one of the following two databases.

Movie Database. We consider a simple database that maintains information about movies. The database schema consists of a single database relation: *Movies (mid, title, year, director, genre, language, duration)*. We consider three context parameters as relevant: *accompanying_people*, *mood* and *time_period*. Users express preferences about movies that depend on the values of these context parameters. For instance, users may give a high preference score to movies of the genre *cartoons* when accompanied by their *children* and a lower preference score when accompanied by their *friends*.

Points of Interest Database. The database schema consists of a single database relation with schema: *Points_of_Interest (pid, name, type, location, open-air, hours_of_operation, admission_cost)*. In this example, we consider context parameters *location*, *weather* and *accompanying_people*. For instance, a point of interest of type *zoo* may be a more preferable place to visit than a *brewery* when accompanied by *family* and an *open-air* place like *Acropolis* a better place to visit than a non *open-air museum* when weather is *good*.

3.1.1 Context Model

We model context using a finite set of special-purpose attributes, called *context parameters* (C_i). For a given application X, we define its context environment CE_X as a set of n context parameters $\{C_1, C_2, \ldots, C_n\}$. For instance, the context environment of our movie example is {*accompanying_people, mood, time_period*}, while the context environment of the points of interest example is {*location, weather, accompanying_people*}.

To allow more flexibility in defining preferences, we model context parameters as multidimensional attributes. The dimensions of each parameter are organized in a *hierarchy schema*. We denote the hierarchy schema of a parameter with m dimensions or *levels* as $L = (L_1, \ldots, L_m)$. L_1 is called the lowest or most detailed level of the hierarchy schema and L_m the top one. We define a total order among the levels of each hierarchy schema L such that $L_1 \prec \ldots \prec L_m$ and use the notation $L_i \preceq L_j$ between two levels to mean $L_i \prec L_j$ or $L_i = L_j$. Regarding our running examples, Figure 3.1 depicts the hierarchy schema of *accompanying_people*, *weather*, *time_period*, *location* and *mood*. For instance, *location* has four levels: *city* (L_1), *country* (L_2), *continent* (L_3) and the top level *ALL* (L_4).

Each level L_j of a parameter C_i is associated with a domain of values that we denote by $dom_{L_j}(C_i)$. As usual, a *domain* is an infinitely countable set of values. We require that, for all parameters, the top level L_m has a single value *All*, i.e. $dom_{L_m}(C_i) = \{All\}$. For a context parameter C_i, $dom(C_i) = \cup_{j=1}^{m} dom_{L_j}(C_i)$. A *concept hierarchy* is an instance of a hierarchy schema. Similar to [44], a concept hierarchy of a context parameter C_i with m levels is represented by a tree with m levels with nodes at each level j, $1 \leq j \leq m$, representing values in $dom_{L_j}(C_i)$. The root node (i.e. level m) represents the value *All*. The relationship between the values at the different levels of a concept hierarchy is achieved through the use of a family of ancestor functions $anc_{L_j}^{L_k}$ [121], $1 \leq j < k \leq m$. The functions $anc_{L_j}^{L_{j+1}}$, $1 \leq j < m$, assign each value of the domain of L_j to a value of the domain of L_{j+1}. An edge from a node at level L_j to a node at level L_{j+1} in the concept hierarchy represents that the latter is the ancestor of the former. Given three levels L_j, L_k and L_l, $1 \leq j < k < l \leq m$, the function $anc_{L_j}^{L_l}$ is equal to the composition $anc_{L_j}^{L_k} \circ anc_{L_k}^{L_l}$. Finally, the function $desc_{L_j}^{L_{j+1}}$, $1 \leq j < m$, is the inverse of $anc_{L_j}^{L_{j+1}}$, that is, $desc_{L_j}^{L_{j+1}}(v) = \{x \in dom_{L_j}(C_i) \mid anc_{L_j}^{L_{j+1}}(x) = v\}$.

Figure 3.1 depicts the concept hierarchies of our five example context parameters. Such concept hierarchies may be constructed using for example, the WordNet [93] or other ontologies. In our example, a value of *country* is *Greece*, two *cities* are *Athens* and *Thessaloniki* and $anc_{L_1}^{L_2}(Athens) = Greece$.

A *context state* corresponds to an assignment of values from their domains to context parameters. In particular, a context state cs is a n-tuple of the form $(c_1, c_2, \ldots, c_n)$, where $c_i \in dom(C_i)$. For instance, a context state in our movie example may be (*friends, good, holidays*) or (*friends, All, summer_holidays*). The set of all possible context states called *world* CW, is the Cartesian product of the domains of the context parameters: $CW = dom(C_1) \times dom(C_2) \times \cdots \times dom(C_n)$.

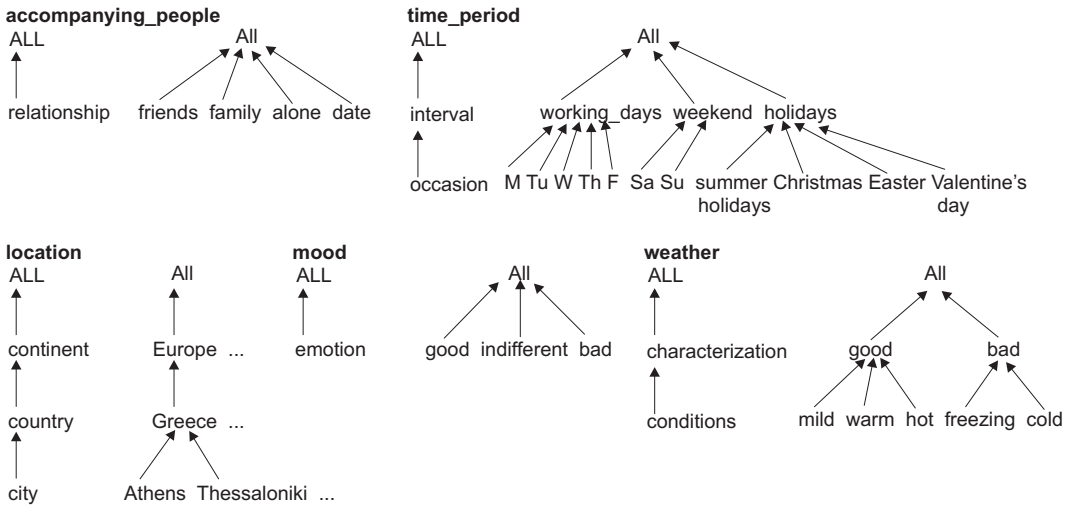


Figure 3.1: Hierarchy schema and concept hierarchy of *accompanying_people*, *weather*, *time_period*, *location* and *mood*.

3.1.2 Context Descriptors

Users express conditions regarding the values of a context parameter through *context descriptors*. Specifically, a single attribute context descriptor is a specification that a user can make for a particular context parameter.

Definition 3.1 (Single attribute context descriptor)**.** A single attribute context descriptor $cod(C_i)$ of a context parameter C_i is an expression of the form $C_i \in \{v_1, \ldots, v_m\}$, where $v_k \in dom(C_i)$, $1 \leq k \leq m$.

For example, for the context parameter *time_period*, a single attribute context descriptor can be *time_period* $\in$ {*Christmas*} or *time_period* $\in$ {*Christmas, Easter, summer_holidays*}. Given a single attribute context descriptor $cod(C_i)$ of a context parameter C_i with $C_i \in \{v_1, \ldots, v_m\}$, its context is a finite set of values, denoted $Context(cod(C_i))$ $= \{v_1, \ldots, v_m\}$.

A user can specify context states through multi-attribute context descriptors that combine single attribute ones.

Definition 3.2 (Multi-attribute context descriptor)**.** A multi-attribute context descriptor *cod* is a formula $cod(C_{i_1}) \wedge cod(C_{i_2}) \wedge \ldots \wedge cod(C_{i_k})$, where each C_{i_j}, $1 \leq j \leq k$, is a context parameter and there is at most one single attribute context descriptor per context parameter C_{i_j}.

Given a set of context parameters $C_1, \ldots, C_n$, a multi-attribute context descriptor specifies a set of context states. These states are computed by taking the Cartesian product of the contexts of all the single attribute context descriptors that appear in the descriptor. When the multi-attribute context descriptor does not contain descriptors for all context parameters, we assume that the absent context parameters have irrelevant values. In particular, if a context parameter C_i is missing from a multi-attribute descriptor, we assume the implicit condition $C_i \in \{All\}$ to be part of the descriptor. Formally:

Definition 3.3 (Context of a multi-attribute context descriptor)**.** Assume a set of context parameters $C_1, \ldots, C_n$ and a multi-attribute context descriptor $cod = cod(C_{i_1}) \wedge \cdots \wedge cod(C_{i_k})$, $1 \leq k \leq n$. Without loss of generality, we assume that the parameters without a context descriptor are the last $n - k$ ones. The context states of a multi-attribute context descriptor, called $Context(cod)$, are defined as: $Context(cod(C_{i_1})) \times \ldots \times Context(cod(C_{i_k})) \times \{All\} \times \ldots \times \{All\}$.

Suppose for instance, for the movie example, the multi-attribute context descriptor (*accompanying_people* $\in$ {*friends, family*} $\wedge$ *time_period* $\in$ {*summer_holidays*}). This descriptor defines the following two context states: (*friends, All, summer_holidays*) and (*family, All, summer_holidays*).

3.1.3 Contextual Preference Model

Contextual preferences have two parts: one specifies the preference and the other one the context states under which the preference holds. For specifying these context states, we use context descriptors as introduced in the previous section.

Regarding preference specification, there are in general, two different approaches: a quantitative and a qualitative one. In the *quantitative approach* (e.g. [11]), preferences are expressed indirectly by using scoring functions that associate a numeric score with each tuple of the query answer. In the *qualitative approach* (e.g. [32, 74]), preferences between tuples in the query answer are specified directly, typically using binary preference relations. Our context model can be used for extending both quantitative and qualitative approaches. Here, we use a simple quantitative preference model to demonstrate the basic issues underlying contextualization.

In particular, users express their preference for specific database instances by providing a numeric score which is a real number between 0 and 1. This score expresses their degree of interest. Value 1 indicates extreme interest, while value 0 indicates no interest. Interest is expressed for specific values of non context attributes of a database relation, for instance, for the various attributes (e.g. *genre, language*) of our movie database relation. This is similar to the general quantitative framework of [11]. Thus, formally:

Definition 3.4 (Contextual Preference)**.** Given a database schema $R(A_1, A_2, \ldots, A_d)$, a *contextual preference* p on R is a triple (*cod*, $Pred$, *score*), where

1. *cod* is a multi-attribute context descriptor,
2. $Pred$ is a predicate of the form $A_{i_1}\theta_{i_1}a_{i_1} \wedge A_{i_2}\ \theta_{i_2}\ a_{i_2} \wedge \ldots \wedge A_{i_k}\theta_{i_k}\ a_{i_k}$ that specifies conditions θ_{i_j} on the values $a_{i_j} \in dom(A_{i_j})$ of attributes A_{i_j}, $1 \leq i_j \leq d$, of R and
3. *score* is a real number between 0 and 1.

The meaning of such a contextual preference is that in the set of context states specified by *cod*, all database tuples (instances) that satisfy the predicate $Pred$ are assigned the indicated interest *score*. In our running examples, we assume that $\theta \in \{=, <, >, \leq, \geq, \neq\}$

mid	*title*	*year*	*director*	*genre*	*language*	*duration*
t_1	Casablanca	1942	Curtiz	Drama	English	102
t_2	Psycho	1960	Hitchcock	Horror	English	109
t_3	Schindler's List	1993	Spielberg	Drama	English	195

Figure 3.2: Movie database instance example.

for the numerical database attributes and $\theta \in \{=, \neq\}$ for the remaining ones. For instance, assume the movie relation of Figure 3.2 of our movie database example. A user can express the fact that when in a *bad* mood, *alone* at a *weekend* or during a *holiday*, he prefers *horror* movies with interest score 0.8 by specifying the preference: $((accompanying\_people \in \{alone\} \wedge mood \in \{bad\} \wedge time\_period \in \{weekend,\ holidays\}),\ genre = horror,\ 0.8)$.

By using multi-attribute context descriptors, users can express preferences that depend on more than one context parameter, for instance, on *Valentine*'s day, one may like to watch *romance* movies when on a *date*, but not when out with *friends*. Furthermore, hierarchies allow the specification of preferences at various levels of detail. For instance, one can specify preferences at the *country*, *city* or both levels.

In the following, we call *profile* P the set of all contextual preferences that hold for an application. The context $Context(P)$ of a profile P is the union of the contexts of all context descriptors that appear in P, that is, $Context(P) = \cup_i Context(cod_i)$, for each $(cod_i, Pred_i, score_i) \in P$.

3.2 Context Resolution

In this section, we focus on determining which of the contextual preferences to use for personalizing a query. We call this problem *context resolution*.

3.2.1 Contextual Queries

We consider queries augmented with information regarding context. We call such queries *contextual queries*. Formally:

Definition 3.5 (Contextual Query)**.** A contextual query CQ is a query enhanced with a multi-attribute context descriptor cod^{CQ}.

The context descriptor may be postulated by the application or be explicitly provided by the users as part of their queries. Typically, in the first case, the context implicitly associated with a contextual query corresponds to the current context, that is, the context surrounding the user at the time of the submission of the query. To capture the current context, context-aware applications use various devices, such as temperature sensors or GPS-enabled devices for location. Methods for capturing context are beyond the scope of this work.

Besides this implicit context, we also envision queries that are explicitly augmented with multi-attribute context descriptors by the users issuing them. For example, such descriptors may correspond to exploratory queries of the form: what is a good film to watch with my *family* this *Christmas* or what are the interesting points not to be missed when I visit *Athens* with my *friends* next summer.

The context associated with a query may correspond to a single context state, where each context parameter takes a specific value from its most detailed domain. However, in some cases, it may be only possible to specify the current context using rough values, for example, when the context values are provided by sensor devices with limited accuracy. In such cases, a context parameter may take a single value from a higher level of the hierarchy or even more than one value.

Now, given a contextual query, the issue is which preferences from the profile should be used to personalize the query. In this work, we focus on context related issues, that is, on selecting those contextual preferences from the profile whose context states match best the context states specified in the query. We call this problem *context resolution*. Once the appropriate preferences are selected, the query can be extended to take them into account, as in the case of non-contextual preferences (e.g. [83, 77]).

Context resolution is studied in two steps: (a) identifying all candidate context states in the profile that encompass the query states and (b) selecting the most appropriate states among these candidates. The first subproblem is resolved through the notion of the *cover partial order* between states that relates context states expressed at different hierarchy levels. To handle the second subproblem, we propose two distance metrics that capture similarity between context states to allow choosing among the candidate states those that are most similar to the query ones.

3.2.2 The Cover Relation

Let us first consider a simple example related to the movie database. Assume a contextual query CQ enhanced with the context descriptor $cod^{CQ} = (accompanying\_people \in \{friends\} \wedge mood \in \{good\} \wedge time\_period \in \{summer\_holidays\})$. If a preference with exactly the same context descriptor exists in the profile P, context resolution is straightforward and this preference is selected. Assume now, that this is not the case. Instead, profile P consists of three preferences: $p_1 = ((accompanying\_people \in \{friends\} \wedge mood \in \{good\} \wedge time\_period \in \{holidays\}), Pred_1, score_1)$ and $p_2 = ((accompanying\_people \in \{friends\} \wedge mood \in \{good\} \wedge time\_period \in \{All\}), Pred_2, score_2)$ and $p_3 = ((accompanying\_people \in \{friends\} \wedge mood \in \{good\} \wedge time\_period \in \{Working\_days\}), Pred_3, score_3)$. Intuitively, in the absence of an exact match, we would like to use those preferences in P whose context descriptor is more general than the query descriptor, in the sense that its context "covers" that of the query:

Definition 3.6 (Covering context state)**.** A context state $cs^1 = (c_1^1, c_2^1, \ldots, c_n^1) \in CW$ covers a context state $cs^2 = (c_1^2, c_2^2, \ldots, c_n^2) \in CW$, if and only if, $\forall\ k,\ 1 \leq k \leq n$, $c_k^1 = c_k^2$ or $c_k^1 = anc_{L_i}^{L_j}(c_k^2)$ for some levels $L_i \prec L_j$.

In the example above, the context states of p_1 and p_2 cover that of q, whereas those of p_3 do not. It can be shown that the cover relationship impose a partial order among context states.

Theorem 3.1. *The cover relationship among context states is a partial order relationship.*

Proof. We must show that the cover relationship is (i) reflexive (i.e. cs covers cs), (ii) antisymmetric (if cs^1 covers cs^2 and cs^2 covers cs^1, then $cs^1 = cs^2$) and (iii) transitive (if cs^1 covers cs^2 and cs^2 covers cs^3, then cs^1 covers cs^3).

(i) Reflexivity is straightforward.

(ii) Assume for the purpose of contradiction that the antisymmetric property does not hold. In this case, there is a certain parameter C_k for which $c_k^1 = anc_{L_i}^{L_j}(c_k^2)$ and $c_k^2 = anc_{L_j}^{L_i}(c_k^1)$. But this cannot happen due to the total order of levels in a hierarchy.

(iii) Assume that cs^1 covers cs^2 (1) and cs^2 covers cs^3 (2). From (1), $\forall k$, $1 \leq k \leq n$, $c_k^1 = c_k^2$ or $c_k^1 = anc_{L_i}^{L_j}(c_k^2)$, $L_i \prec L_j$ (3). Respectively, from (2), $\forall k$, $1 \leq k \leq n$, $c_k^2 = c_k^3$ or $c_k^2 = anc_{L_i}^{L_j}(c_k^3)$, $L_i \prec L_j$ (4). Therefore, from (3), (4), we get that, $\forall k$, $1 \leq k \leq n$, $c_k^1 = c_k^3$ or $c_k^1 = anc_{L_i}^{L_j}(c_k^3)$, $L_i \prec L_j$, that is, cs^1 covers cs^3.

∎

Going back to our example, although the context states of both p_1 and p_2 cover those of the query CQ, p_1 is more closely related to the descriptor of the query and it is the one that should be used. Next, we formalize this notion of the most specific or exact covering state.

Definition 3.7 (Exact covering state)**.** Let P be a profile and cs^1 a context state. We say that a context state $cs^2 \in Context(P)$ is an exact cover for cs^1 in P, if and only if:

(i) cs^2 covers cs^1 and

(ii) $\nexists\ cs^3 \in Context(P)$, $cs^3 \neq cs^2$, such that, cs^2 *covers* cs^3 and cs^3 *covers* cs^1.

Note that there may be more than one exact cover. For example, consider again the previous query context descriptor cod^{CQ} and assume now that P includes a fourth preference, $p_4 = ((accompanying\_people \in \{friends\} \wedge mood \in \{All\} \wedge time\_period \in \{summer\_holidays\}), Pred_4, score_4)$. The states of both p_1 and p_4 in P satisfy the first condition of Def. 3.7, but none of them covers the other.

We can now provide a formal definition for context resolution:

Definition 3.8 (Context Resolution)**.** Given a profile P and a contextual query CQ, context resolution refers to identifying a set $RS \subseteq Context(P)$ of context states such that (a) for each context state $cs^q \in Context(cod^{CQ})$, there exists at least one context state cs^p in RS such that cs^p is an exact cover of cs^q in P and (b) cs^p belongs to RS only if there is a $cs^q \in Context(cod^{CQ})$ for which cs^p is an exact cover in P.

Having identified these states, we use the contextual preferences associated with the corresponding descriptors for personalizing the query.

Two additional issues need to be addressed. One is what happens if for some context state in the context of the query, there is no context state in profile P that covers it. We assume that there is always a default preference with an empty context descriptor, that is, a context descriptor that corresponds to the context state $(All, All, \ldots, All)$. The other issue is what happens when a context state of the query has more than one exact covering state in P. The definition above just specifies that *at least one* of them needs to be used. There are many approaches to resolving this issue. One is to let the user decide. In this case, all exact covering preferences are presented to the users and they decide which ones to use. In the next section, we provide a systematic way to choose which of the profile states that are exact covering states of a query state to use by defining distances among context states.

3.2.3 Context State Similarity

To select the most appropriate among a number of exact covering states, we introduce a similarity (or distance) metric between context states. The motivation is to choose the most specific among the candidate states, that is, the states defined in detailed hierarchy levels. We define first the level of a state as follows.

Definition 3.9 (Levels of a state)**.** Given a context state $cs = (c_1, c_2, \ldots, c_n)$, the hierarchy levels that correspond to this state are $levels(cs) = [L_{j_1}, L_{j_2}, \ldots, L_{j_n}]$ such that $c_i \in dom_{L_{j_i}}(C_i)$, $i = 1, \ldots, n$.

The distance between two levels is defined as their path distance in their hierarchy schema. Formally:

Definition 3.10 (Level distance)**.** Given two levels L_i and L_j of a hierarchy schema L, the level distance $dist_H(L_i, L_j)$ is equal to the number of edges that connect L_i and L_j in L.

Having defined the distance between two levels, we can now define the level-based distance between two states.

Definition 3.11 (Hierarchy state distance)**.** Given two states $cs^1 = (c_1^1, c_2^1, \ldots, c_n^1)$ and $cs^2 = (c_1^2, c_2^2, \ldots, c_n^2)$, the hierarchy state distance $dist_H(cs^1, cs^2)$ is defined as:

$$dist_H(cs^1, cs^2) = \sum_{i=1}^{n} dist_H(L_i^1, L_i^2).$$

For example, the hierarchy state distance of context states $cs^1 =$ (*friends, good, summer_holidays*) and $cs^2 =$ (*friends, good, holidays*) is equal to: $dist_H(cs^1, cs^2) = 1$.

We show next, that the hierarchy state distance produces an ordering of states that is compatible with the cover partial order in the sense expressed by the following property.

Property 3.1. *Assume a state* $cs^1 = (c_1^1, c_2^1, \ldots, c_n^1)$. *For any two different states* $cs^2 = (c_1^2, c_2^2, \ldots, c_n^2)$ *and* $cs^3 = (c_1^3, c_2^3, \ldots, c_n^3)$, $cs^2 \neq cs^3$, *such that* cs^2 *covers* cs^1 *and* cs^3 *covers* cs^1, *if* cs^3 *covers* cs^2, *then* $dist_H(cs^1, cs^3) > dist_H(cs^1, cs^2)$.

Proof. Let $level(cs^1) = [L_1^1, L_2^1, \ldots, L_n^1]$, $level(cs^2) = [L_1^2, L_2^2, \ldots, L_n^2]$ and $level(cs^3) = [L_1^3, L_2^3, \ldots, L_n^3]$. From Def. 3.6, since cs^2 covers cs^1 and the fact that the level of any ancestor of c_i is larger than the level of c_i, it holds $L_i^2 \succeq L_i^1$, $\forall i$, $1 \leq i \leq n$ (1). Similarly, since cs^3 covers cs^1, it holds $L_i^3 \succeq L_i^1$, $\forall i$, $1 \leq i \leq n$ (2) and, since cs^3 covers cs^2, it holds $L_i^3 \succeq L_i^2$, $\forall i$, $1 \leq i \leq n$ (3). From (1), (2) and (3), we get $L_i^3 \succeq L_i^2 \succeq L_i^1$, $\forall i$, $1 \leq i \leq n$ (4). Since $cs^2 \neq cs^3$, for at least one j, $1 \leq j \leq n$, it holds $L_j^3 \succ L_j^2$ (5). Thus, from (4), (5) and Def. 3.11, it holds that $dist_H(cs^1, cs^3) > dist_H(cs^1, cs^2)$. ∎

Property 3.1 states that between two covering states, the exact covering state is the one with the smallest hierarchy distance.

However, in some cases, the context state with the minimum hierarchy distance is not unique. For instance, assume that we want to select the most similar context states to cs^1 = (*friends, good, summer_holidays*) between the states cs^2 = (*friends, good, holidays*) and cs^3 = (*friends, All, summer_holidays*). For these states, $dist_H(cs^1, cs^2) = dist_H(cs^1, cs^3)$ = 1. To resolve such ties, again we choose those states that are more specific but now in terms of the values of the detailed (lowest) level of the hierarchy that they include. The motivation is that context values that have few detailed values as descendants are more specific than those that have more such values. Thus, we consider the set of descendants of each value of a state. For two values of two states corresponding to the same context parameter, we measure the fraction of the intersection of their corresponding lowest level value sets over the union of these two sets and consider as a better match, the "smallest" state in terms of cardinality. Formally, this is expressed through the Jaccard distance of two values c_1 and c_2 of the same hierarchy schema L defined as follows:

Definition 3.12 (Jaccard distance)**.** The Jaccard distance of two values c_1 and c_2 belonging to levels L_i and L_j of the same hierarchy schema L that has as lowest level the level L_1, is defined as:

$$dist_J(c_1, c_2) = 1 - \frac{|desc_{L_1}^{L_i}(c_1) \bigcap desc_{L_1}^{L_j}(c_2)|}{|desc_{L_1}^{L_i}(c_1) \bigcup desc_{L_1}^{L_j}(c_2)|}.$$

It is easy to show that values at higher levels in the hierarchy have larger Jaccard distances than their descendants at lower levels, as the following property states:

Property 3.2. *Assume three values* c_1, c_2, c_3 *defined at different levels, say* L_2, L_3, L_4, *with* $L_2 \prec L_3 \prec L_4$, *of the same hierarchy having* L_1 *as the most detailed level such that* $c_3 = anc_{L_3}^{L_4}(c_2) = anc_{L_2}^{L_4}(c_1)$ *and* $c_2 = anc_{L_2}^{L_3}(c_1)$. *Then,* $dist_J(c_1, c_3) \geq dist_J(c_1, c_2)$.

Proof. By definition,
$dist_J(c_1, c_2) = 1 - \frac{|desc_{L_1}^{L_2}(c_1) \bigcap desc_{L_1}^{L_3}(c_2)|}{|desc_{L_1}^{L_2}(c_1) \bigcup desc_{L_1}^{L_3}(c_2)|}$ (1)
and

$dist_J(c_1, c_3) = 1 - \frac{|desc_{L_1}^{L_2}(c_1) \bigcap desc_{L_1}^{L_4}(c_3)|}{|desc_{L_1}^{L_2}(c_1) \bigcup desc_{L_1}^{L_4}(c_3)|}$ (2).
In both fractions, the numerator reduces to $desc_{L_1}^{L_2}(c_1)$, clearly due to the transitivity property of the ancestor functions. The denominator of the first fraction is $desc_{L_1}^{L_3}(c_2)$, whereas the denominator of the second fraction is $desc_{L_1}^{L_4}(c_3) \supseteq desc_{L_1}^{L_3}(c_2)$, again due to the transitivity property of the ancestor function (i.e. all descendants of c_2 at the detailed level are also descendants of c_3). Therefore $dist_J(c_1, c_3) \geq dist_J(c_1, c_2)$. ∎

The Jaccard state distance between two states is defined as:

Definition 3.13 (Jaccard state distance)**.** Given two states $cs^1 = (c_1^1, c_2^1, \ldots, c_n^1)$ and $cs^2 = (c_1^2, c_2^2, \ldots, c_n^2)$, the Jaccard state distance $dist_J(cs^1, cs^2)$ is defined as:
$$dist_J(cs^1, cs^2) = \sum_{i=1}^{n} dist_J(c_i^1, c_i^2).$$

For example, the Jaccard distance of context states cs^1 = (*friends, good, summer_holidays*) and cs^2 = (*friends, good, holidays*) is equal to: $dist_J(cs^1, cs^2) = 3/4$. Now, returning to our previous example for cs^1 = (*friends, good, summer_holidays*) and the two candidates states, cs^2 = (*friends, good, holidays*) and cs^3 = (*friends, All, summer_holidays*), with the same hierarchy state distance, it holds that $dist_J(cs^1, cs^2) = 3/4$ and $dist_J(cs^1, cs^3) = 2/3$. Therefore, the most similar state to cs^1 is state cs^3.

It is easy to prove a property similar to Property 3.1, that is:

Property 3.3. *Assume a state $cs^1 = (c_1^1, c_2^1, \ldots, c_n^1)$. For any two different states $cs^2 = (c_1^2, c_2^2, \ldots, c_n^2)$ and $cs^3 = (c_1^3, c_2^3, \ldots, c_n^3)$, $cs^2 \neq cs^3$, that both cover cs^1, that is, cs^2 covers cs^1 and cs^3 covers cs^1, if cs^3 covers cs^2, then $dist_J(cs^1, cs^3) > dist_J(cs^1, cs^2)$.*

Proof. Let $level(cs^1) = [L_1^1, L_2^1, \ldots, L_n^1]$, $level(cs^2) = [L_1^2, L_2^2, \ldots, L_n^2]$ and $level(cs^3) = [L_1^3, L_2^3, \ldots, L_n^3]$. From the proof of Property 3.1, we have that $L_i^3 \succeq L_i^2 \succeq L_i^1$, $\forall i$, $1 \leq i \leq n$. From Def. 3.12, $\forall\ c_i^1, c_i^2, c_i^3$, $1 \leq i \leq n$, we get that $dist_J(c_i^1, c_i^3) > dist_J(c_i^1, c_i^2)$ (1), because that fraction becomes smaller as the context values belong to higher hierarchy levels. From (1) and Def. 3.13, we get that $dist_J(cs^1, cs^3) > dist_J(cs^1, cs^2)$. ∎

To summarize, when there are more than one exact covering context state in the profile, we select the one that has the smallest hierarchy state distance from the query state. If more than one such state exists, we order them using their Jaccard state distance from the query state.

3.2.4 Scores based on Predicate Subsumption

The result of context resolution is the set of context states cs^j in the profile P that are the most similar to the query context. Next, the preferences $(cod_i, Pred_i, score_i) \in P$ such that $cs^j \in Context(cod_i)$ are selected and applied to produce a score for the results of the query.

In general, more than one of the selected preferences may be applicable to a specific database tuple t in the result r. In other words, a tuple t may satisfy the predicate part

of more than one of the selected preferences. We shall use the notation $Pred[t]$ to denote that tuple t satisfies predicate $Pred$. Let us consider first the special case in which the selected predicates are related by subsumption. Given two predicates $Pred_1$ and $Pred_2$, we say that $Pred_1$ subsumes $Pred_2$, if and only if, $\forall\ t \in r$, $Pred_1[t] \Rightarrow Pred_2[t]$. In this case, we say that $Pred_1$ is *more specific* than $Pred_2$. When a tuple t satisfies predicates that one subsumes the other, to compute a score for t, we consider only the preferences with the most specific predicates because these are considered as used by the users to specialize or refine the more general ones. In all other cases, we use the preference with the highest score, considering preferences to be indicators of positive interest.

Definition 3.14 (Tuple Score)**.** Let P be a profile, cs a context state and $t \in r$ a tuple. Let $P' \subseteq P$ be the set of preferences $p_i = (cod_i,\ Pred_i,\ score_i)$ such that $cs \in Context(cod_i)$, $Pred_i[t]$ holds and $\nexists\ p_j = (cod_j,\ Pred_j,\ score_j) \in P'$ such that $cs \in Context(cod_j)$, $Pred_j[t]$ holds and $Pred_j$ subsumes $Pred_i$. The score of t in cs is: $score(t, cs) = max_{p_i \in P'} score_i$.

For example, assume the movie relation of Figure 3.2 and a profile with the following simple preferences: $p_1 = ((accompanying\_people \in \{friends\}),\ genre = horror,\ 0.8)$, $p_2 = ((accompanying\_people \in \{friends\}),\ director = Hitscock,\ 0.7)$, $p_3 = ((accompanying\_people \in \{alone\}),\ genre = drama,\ 0.9)$, $p_4 = ((accompanying\_people \in \{alone\}),\ (genre = drama \wedge director = Spielberg),\ 0.5)$. In the context state (*friends, All, All*), tuple t_2 satisfies the predicates of both preferences p_1 and p_2. Similarly, in context state (*alone, All, All*), tuple t_3 satisfies the predicates of both preferences p_3 and p_4. In the first case, none of the two predicates subsumes the other and the score for t_2 is the maximum of the two scores, namely 0.8. Under context state (*alone, All, All*), the predicate of p_4 subsumes the predicate of p_3 and so, t_3 has score 0.5. The motivation is that the user has assigned to *drama* movies in general, score 0.9 and to *drama* movies directed by *Spielberg* in particular, score 0.5. Tuple t_3 belongs to the second category and thus, it is assigned the corresponding score.

Definition 3.14 specifies how to compute the score of a tuple under a specific context state. However, the result of context resolution for a query CQ may include more than one context state.

Definition 3.15 (Aggregate Tuple Score)**.** Let P be a profile, $CS \subseteq Context(P)$ be a set of context states and $t \in r$ a tuple. The score of t in CS is: $score(t, CS) = max_{cs \in CS} score(t, cs)$.

It is straightforward (by Definition 3.15) that:

Property 3.4. *Let cs be a context state and CS a set of context states. If cs ∈ CS, then for any t ∈ r, score(t, CS) ≥ score(t, cs).*

This means that the score of a tuple computed using a set of context states is no less than the score of the tuple computed using any of the context states belonging to this set.

Other ways of aggregating scores besides choosing the highest score are possible. Our goal is not to miss any highly preferred tuple in any of the matching context states, thus, high scores overwrite lower ones. Of course, one can argue for other interpretations; our context resolution procedure and the related algorithms are still applicable.

3.3 Data Structures and Algorithms

Given a profile P and a contextual query CQ with descriptor cod^{CQ}, our goal is to determine which of the contextual preferences in P to use for personalizing CQ. To this end, we need to locate the appropriate states in $Context(P)$ based on their similarity to the query context $Context(cod^{CQ})$. One way to achieve this is by sequentially scanning all context states of all preferences and retrieving those with the smallest distances from each query state. To improve response time and storage overheads, we consider indexing the preferences in P based on the context states in $Context(P)$.

3.3.1 Preference Graph

We introduce a graph representation of preferences that exploits the cover relation between context states to organize user preferences. Specifically, the *preference graph* of profile P is a directed acyclic graph in which there is one node v_i for each context state $cs_i \in Context(P)$ and an edge from v_i to v_j, if and only if, cs_i is an exact cover for cs_j. In addition, each node v_i in the graph is associated with the set of predicates and the related interest scores of all preferences that include in their context descriptor the state cs_i. This is called the score set of the state. Formally, the *score set* of a context state cs is the set $W_{cs} = \{(Pred_i, score_i) \mid (cod_i, Pred_i, score_i) \in P \text{ and } cs \in Context(cod_i)\}$. Thus:

Definition 3.16 (Preference Graph)**.** Given a profile P, the preference graph $PG_P = (V_P, E_P)$ of P is a directed acyclic graph, where for each context state $cs_i \in Context(P)$, there exists a node v_i, $v_i \in V_P$, of the form (cs_i, W_{cs_i}), where W_{cs_i} is the score set of cs_i. Given two nodes $v_i, v_j \in V_P$, an edge $(v_i, v_j) \in E_P$, if and only if, cs_i is an exact cover of cs_j.

For example, for the movie database and the profile with context states as shown in Figure 3.3(a), the preference graph depicted in Figure 3.3(b) is constructed.

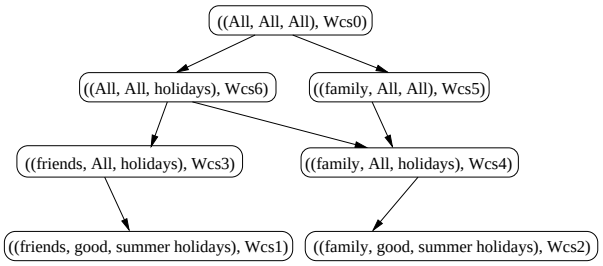
Note that the preference graph is acyclic, because the cover relationship among context states induces a partial order among them (Theorem 3.1). Note also that, given a set of n context parameters $C_1, C_2, \ldots C_n$ with $h_1, h_2, \ldots, h_n$ hierarchy levels, the maximum path length between any two nodes in the graph, i.e. the maximum number of edges that connect them, is equal to $h_1 + h_2 + \ldots + h_n - (n + 1)$.

Context Resolution. Given a contextual query CQ with context descriptor cod^{CQ}, for each context state $cs \in Context(cod^{CQ})$, we search the preference graph for states that match it. Specifically, we traverse the preference graph top-down starting from the nodes

(a) Context States with Score Sets

cs_1: ((friends, good, summer holidays), W_{cs_1})
cs_2: ((family, good, summer holidays), W_{cs_2})
cs_3: ((friends, All, holidays), W_{cs_3})
cs_4: ((family, All, holidays), W_{cs_4})
cs_5: ((family, All, All), W_{cs_5})
cs_6: ((All, All, holidays), W_{cs_6})

(b) Preference Graph



(c) Profile Tree

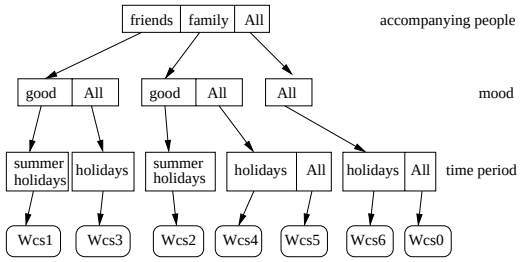


Figure 3.3: (a) A set of context states and an instance of (b) a preference graph and (c) a profile tree.

with no incoming edges and following their outgoing edges. Search at a node stops when there no outgoing edges or if the state of the node does not cover the query state cs, thus some search paths are pruned.

Specifically, the *PG_Resolution Algorithm* (Algorithm 8) returns results of the form (W_{cs_i}, *distance*) that correspond to nodes v_i whose context state cs_i is an exact cover of the query context state cs. The returned distance values refer to the distances between cs and the states cs_i of the preferences of the returned predicates. The correctness of the algorithm is based on the following observation.

Property 3.5. *A state cs_v of a node v is an exact cover of cs, if and only if, cs_v covers cs and (i) v is a leaf node or (ii) v is an internal node and none of its children covers cs.*

This holds, because in both cases there is no other state that is covered by cs_v and covers cs, since if there were one, v should have an outgoing edge to the corresponding node.

If there are more than one exact cover, we select the one with the minimum distance, i.e. the one that differs the least from the query state based on the hierarchy state distance and, in case of ties, the Jaccard state distance. If again there are ties, all states are considered as relevant.

The *PG_Resolution Algorithm* follows a top-down strategy by traversing the graph starting from the nodes with the most general context states and moving towards the nodes with the most specific ones. We can also follow a bottom-up approach and traverse the graph starting from the nodes with the most specific context states up to the nodes with the most general ones. In this case, at each search path, search stops when the first

PG_Resolution Algorithm

Input: A preference graph PG_P of a profile P, the searching context state cs.

Output: A $ResultSet$ of the form (W, distance) characterizing a node whose context state exact covers the searching context state.

```
1: Begin
2: ResultSet = ∅;
3: tmpV_P = ∅;
4: for all nodes v_i ∈ V_P do
5:    if v_i has no incoming edges then
6:       tmpV_P = tmpV_P ∪ {v_i};
7: while tmpV_P not empty do
8:    for all v_i ∈ tmpV_P do
9:       if cs_i covers cs then
10:         if v_i has no outgoing edges then
11:            ResultSet = ResultSet ∪ {(W_{cs_i}, dist(cs_i, cs))};
12:         else
13:            if ∀ v_j s. t. (v_i, v_j) ∈ E_P, cs_j does not cover cs then
14:               ResultSet = ResultSet ∪ {(W_{cs_i}, dist(cs_i, cs))};
15:            else
16:               for all v_q s. t. (v_i, v_q) ∈ E_P and v_q unmarked do
17:                  tmpV_P = tmpV_P ∪ {v_q};
18:                  mark v_q;
19:      tmpV_P = tmpV_P − {v_i};
20: End
```

Algorithm 8: PG_Resolution Algorithm

node that covers the query state is met. In general, the bottom-up traversal visits more nodes than the top-down one, since the number of nodes at the lower levels of the graph tends to be much larger than that of the nodes higher up.

Intuitively, the bottom-up traversal is expected to outperform the top-down one only for query states with specific context values, that is, for query states with context values that belong to the lower levels of their corresponding hierarchies. To take advantage of this simple observation, we use the following heuristic for selecting the appropriate type of traversal.

We associate with each query state cs a level score ls equal to the average value of the hierarchy levels of the context values in cs. Similarly, we compute a level score lp for the preference graph as follows. Let n be the number of nodes in the graph and ls_i the level score of the context state of node n_i, then, $lp = (\sum_{i=1}^{n} ls_i)/n$. We expect that if a query state cs has level score greater (resp. smaller) than the level score of the graph, the matching state of cs will appear higher (resp. lower) in the graph and so, the top-down (resp. bottom-up) approach is selected.

3.3.2 Profile Tree

The *profile tree* explores the repetition of context values in context states by using a prefix-based approach for storing the context states in $Context(P)$ of the profile P. Each context state cs in $Context(P)$ corresponds to a single root-to-leaf path in the tree. Assume that the context environment CE_X has n context parameters $\{C_1, C_2, \ldots, C_n\}$. The profile tree for P has n+1 levels. Each one of the first n levels corresponds to one of the context parameters and the last one is the level of the leaf nodes. For simplicity, assume that context parameter C_i is mapped to level i of the tree. At level C_1, there is a single root node. Each non-leaf node at level k $(1 \leq k \leq n)$ contains cells of the form $[key, pointer]$, where key corresponds to some value of the parameter C_k that appeared in some state cs in $Context(P)$. No two cells within the same node contain the same key value. The pointer of each cell of the nodes at level k, $(1 \leq k < n)$, points to the node at the next lower level (level $k+1$) containing all the distinct values of the next context parameter (parameter C_{k+1}) that appeared in the same context state cs with key. The pointers of cells of the nodes at level n point to leaf nodes. Each leaf node contains the score set W_{cs} of the state cs that corresponds to the path leading to it.

For example, regarding the movie database, the profile tree for the context states of Figure 3.3(a) is depicted in Figure 3.3(c).

The size of the tree depends on which context parameter is assigned to each level. Let m_i, $1 \leq i \leq n$, be the cardinality of the domain of parameter C_i, then the maximum number of cells is $m_1 \times (1 + m_2 \times (1 + \ldots (1 + m_n)))$. The above number is as small as possible when $m_1 \leq m_2 \leq \cdots \leq m_n$, thus, in general, it is better to place context parameters with domains with higher cardinalities lower in the profile tree.

Context Resolution. We describe next how the profile tree is used for context resolution. Given a contextual query CQ with a context descriptor cod^{CQ}, for each query context state $cs = (c_1, c_2, \ldots, c_n) \in Context(cod^{CQ})$, we search the profile tree for a state that matches it. If there is a state that exactly matches it, that is a state $(c_1, c_2, \ldots, c_n)$, then, the associated preference is returned to the user. Note that this state is easily located by a single depth-first-search traversal of the profile tree. Starting from the root of the tree (level 1), at each level i, we follow the pointer associated with $key = c_i$. If such a state does not exist, we search for a state cs' that best covers cs. If more than one such state exists, we select the one with the smallest distance using the hierarchy distance and, in case of ties, the Jaccard distance.

Given a profile tree whose root node is R_P, the *PT_Resolution Algorithm* (Algorithm 9) descends the profile tree starting from the root node in a breadth first manner. It maintains all paths whose context state is either the same or covers the query context state. Each candidate path counts the distance from the query path. At first we invoke $PT\_Resolution(R_P, \{c_1, c_2, \ldots, c_n\}, 0)$. At the end of the execution of this call, we sort all the results on the basis of their distances and select the one with the minimum distance, i.e. the one that differs the least from the searched path based on the two distances.

Clearly the last step can be easily replaced by a simple runtime check that keeps the

PT_Resolution Algorithm

Input: A node R_P of the profile tree, the query context state $cs = (c_1, c_2, \ldots, c_n)$, the current distance of each candidate path.

Output: A *ResultSet* of the form (W, distance) characterizing a candidate path whose context state is either the same or covers cs.

1: **Begin**
2: **if** R_P is a non leaf node **then**
3: **for** all $x \in R_P$ such that $x = c_i$ or $x = anc_{L_i}^{L_j}(c_i)$ **do**
4: $PT\_Resolution(x \rightarrow child, \{c_{i+1}, \ldots, c_n\}, dist(x, c_i) + distance)$;
5: **else if** R_P is a leaf node containing the score set W_{cs_r} **then**
6: $ResultSet = ResultSet \cup \{(W_{cs_r}, distance)\}$;
7: **End**

Algorithm 9: PT_Resolution Algorithm

current closest leaf if its distance is smaller than the one currently tested. In particular, to reduce the number of candidate paths maintained, we could prune those under construction paths for which there is at least another sub-path that has smaller distance to the searched one, independently of the rest of the values of the path. Assume a query context state $cs = (c_1, \ldots, c_i, c_{i+1}, \ldots, c_n)$. Assume further, that at level i of the tree, we have two candidate sub-paths with values $sp^1 = (sp_1^1, \ldots, sp_i^1)$ and $sp^2 = (sp_1^2, \ldots, sp_i^2)$. If for the distances between the states $cs^1 = (sp_1^1, \ldots, sp_i^1, c_{i+1}, \ldots, c_n)$ and $cs^2 = (sp_1^2, \ldots, sp_i^2, All, \ldots, All)$ and the query state cs holds that $dist(cs^1, cs) > dist(cs^2, cs)$, then we can safely prune sub-path sp^1. This is because even if, for the rest of the values of sp^1, we could find in the tree values equal to that of the query (best case scenario), its distance from the query state would still be greater than that of sp^2 even if we could not find anything better than *All* for the remaining values of sp^2 (worst case scenario).

We show that the algorithm is correct, i.e. if applied to all context states specified by the context descriptor of the query, it leads to the desired set of states according to Def. 3.7. For each state, the algorithm returns a state that is the most similar to the query one, that is, the one with the smallest distance. By Property 3.1, for the hierarchy distance, and Property 3.3, for the Jaccard distance, it is clear that the state with the smallest distance is the one that exactly covers the query state. Also, the set of context states that are returned, specify a context descriptor. This descriptor covers the descriptor of the query, because each state is expressed by another similar one. Furthermore, the textually described variant can give the exact covering descriptor because for each state we select the exact covering state.

The profile tree is very efficient in the case we are looking only for exact matches of the query state $cs = (c_1, c_2, \ldots, c_n)$. In this case, context resolution requires just a simple root-to-leaf traversal. At each level i, we search for the cell having as key the value c_i and descend to the next level, following the corresponding pointer. Thus, we visit as many

nodes as the height of the profile tree. Note that a sequential scan or a preference graph traversal may require visiting all states in $Context(P)$. For the general case of looking for exact covering states, assume that context parameter C_i has h_i hierarchy levels and $val(C_i)$ distinct values in the profile. Then, the number of cells that are visited for each query state is $val(C_1) + val(C_2) \times h_1 + val(C_3) \times h_2 \times h_1 + \cdots + val(C_n) \times h_{n-1} \times \cdots \times h_1$. Again, for a sequential scan or when the preference graph is used, we may need to check all states.

To speed up context resolution, we exploit the hierarchical nature of context parameters, by adding cross edges, named *hierarchical pointers*, among context values that belong to a specific node. In particular, we link each value with its first ancestor that exists in the node, that is, the value that belongs to the first upper level of the corresponding hierarchy. For instance, for the profile tree of Figure 3.3(c), we add cross edges from *good* to *All* at level *mood* and edges from *holidays* to *All* at level *time period*. Also, the values within each node are sorted according to the hierarchy level to which they belong.

Formally, in the resulting *enhanced profile tree* with n context parameters, each non-leaf node at level k maintains cells of the form $[key, label, pointer, hpointer]$, where key and $pointer$ are defined as in the profile tree and $label$ denotes the level key belongs to. The $hpointer$ field points to a value key' that belongs to the same node with key such that $key' = anc_{L_{key}}^{L_{key'}}(key)$ and there is no key'' such that $key' = anc_{L_{key''}}^{L_{key'}}(key'')$ and $key'' = anc_{L_{key}}^{L_{key''}}(key)$. If there is no such key' value, then $hpointer$ points to $null$. The *EnhancedPT_Resolution Algorithm* (Algorithm 10) uses the *enhanced profile tree* to retrieve the most similar preferences to a contextual query CQ. Here, instead of searching for the more general values of each context value, we just follow the hierarchical pointers of the tree to locate them.

3.3.3 Multi-State Context Resolution

Let cod^{CQ} be the context descriptor of a query and $Q_C = Context(cod^{CQ})$ be the set of context states derived from it. In the previous section, we have used the profile tree and the preference graph to check for each individual cs in Q_C. Here, we propose algorithms that tests for all context states in Q_C.

When the profile tree is used, the context states in Q_C are represented by a data structure similar to the profile tree, that we call *query tree*, so that, there is exactly one path in the tree for each $cs \in Q_C$. Again, there is one level in the query tree for each context parameter. The *QueryCR Algorithm* (Algorithm 11) uses both the profile tree and the query tree. In a breadth first manner, the algorithm searches for pairs of nodes that belong to the same level. Each pair consists of a node of the query tree and a node of the profile tree. Initially, there is one pair of nodes, (R_Q, R_P, 0) (level $i = 1$). For each value of the query node R_Q that is equal to a value of the profile node R_P or belongs to a lower hierarchy level, we create a new pair of nodes ($R_Q \rightarrow child$, $R_P \rightarrow child$, $distance$). These nodes refer to the next level ($i + 1$). After having checked all values of all pairs at a specific level, we examine the pairs of nodes created for the immediately next level and

EnhancedPT_Resolution Algorithm

Input: A node R_P of the *enhanced profile tree*, the query context state $cs = (c_1, c_2, \ldots, c_n)$, the current distance of each candidate path.

Output: A *ResultSet* of the form (W, distance).

1: **Begin**
2: **if** R_P is a non leaf node **then**
3: Find the first $x \in R_P$ such that $x = c_i$ or $x = anc_{L_i}^{L_j}(c_i)$;
4: $EnhancedPT\_Resolution(x \rightarrow child, \{c_{i+1}, \ldots, c_n\}, dist(x, c_i) + distance)$;
5: $y = x \rightarrow hpointer$;
6: **while** $y\ != NULL$ **do**
7: $EnhancedPT\_Resolution(y \rightarrow child, \{c_{i+1}, \ldots, c_n\}, dist(y, c_i) + distance)$;
8: $y = y \rightarrow hpointer$;
9: **else if** R_P is a leaf node containing the score set W_{cs_r} **then**
10: $ResultSet = ResultSet \cup \{(W_{cs_r}, distance)\}$;
11: **End**

Algorithm 10: EnhancedPT_Resolution Algorithm

so on. At level $n + 1$, we retrieve from the profile tree, for each query state, the score set of the path with the most similar state to the query one. Observe that the *QueryCR Algorithm* tests for all query states within a single pass of the profile tree.

We follow a similar approach for the preference graph. In particular, the query context states are represented by a graph G_Q, named *query graph*, similar to the preference graph, i.e. there is a node v in G_Q for each context state $cs \in Q_C$ and an edge from a node v to a node u, if and only if, the state of v is an exact cover of the state of u.

First, we perform a topological sort of the nodes in the query graph G_Q as follows. We start by placing all nodes of G_Q that have no incoming edges in a set, say S_1. Then, we proceed by deleting the nodes in S_1 and their outgoing edges from G_Q and placing the nodes that have no incoming edges in the query graph that results after these deletions in another set, say S_2. This procedure is repeated, until there are no remaining nodes in the query graph. Assume that m sets S_i, $1 \leq i \leq m$, are thus produced. We process the query states using this order. Our approach is based on the following observation. Let a node z with state cs_z be returned as an exact match of a node in S_i, $1 \leq i < m$. None of the predecessors of z in the preference graph can be an exact cover of any node in S_{i+1}. However, context states of nodes that have incoming edges from such predecessors (i.e. belong to a different sub-graph) may be exact covers of the states of nodes in S_{i+1}. To take advantage of this, we start with the set S_1 and search the preference graph to find nodes whose state exactly covers any context state in S_1. Let z be any such node returned for S_1. Then, we remove from the preference graph the incoming edges of z. For finding nodes that exactly cover the states in set S_2, we begin the search process from all nodes with no incoming edges in the updated preference graph. This procedure is repeated for

QueryCR Algorithm

Input: The *profile tree* with root node R_P and n context parameters, the *query tree* with root node R_Q, the current distance of each candidate path.

Output: A *ResultSet* of the form (W, distance) characterizing paths whose context states are the same or similar to the states of the query tree.

SN, SN' sets of pairs of nodes, each pair related with a distance value.
Initially: $SN = \{(R_Q, R_P, 0)\}$, $SN' = \emptyset$

1: **Begin**
2: **for** level $i = 1$ to n **do**
3: **for** each pair $sn \in SN$ with $sn = (q\_node, p\_node, distance)$ **do**
4: $\forall x \in q\_node$
5: $\forall y \in p\_node$
6: **if** $x = y$ or $(y = anc_{L_y}^{L_x}(x))$ **then**
7: **if** $i < n$ **then**
8: $SN' = SN' \cup \{(x \rightarrow child, y \rightarrow child, distance + dist(x, y))\}$;
9: **else if** $i = n$ **then**
10: $W = y \rightarrow child$;
11: $ResultSet = ResultSet \cup \{(W, distance)\}$;
12: $SN = SN'$;
13: $SN' = \emptyset$;
14: **End**

Algorithm 11: QueryCR Algorithm

all sets up to S_m.

3.3.4 Discussion

To summarize, we have proposed two data structures for indexing contextual preferences: the preference graph and the profile tree. The preference graph organizes the context states of a profile by exploring the cover relation among them. A top-down traversal of the graph can be used for an incremental specialization of a given context state, whereas a bottom-up traversal for its incremental relaxation. The profile tree offers a space-efficient representation of context states by taking advantage of the co-occurrence of context values in a profile. It supports exact matches of context states very efficiently through a single root-to-leaf traversal.

Context resolution using either the preference graph or the profile tree returns predicate expressions. These expressions are used to determine the tuples of the query result in the underlying database relation and annotate them with the appropriate scores. It is straightforward, and practically orthogonal to our problem, to perform all the produced expressions as selections of the relational algebra over the underlying relation and rank the results by their score computed based on Definition 3.15.

3.4 Usability Evaluation

The goal of our usability study is to justify the use of contextual preferences. In particular, the objective is to show that for a reasonable effort of specifying contextual preferences, users get more satisfying results than when there are no preferences or when preferences are non-contextual.

We used two databases of different sizes: (a) a points of interest database and (b) a movie database. The points of interest database consists of nearly 1 000 real points of interest of the two largest cities in Greece, namely Athens and Thessaloniki. The context parameters are *accompanying_people*, *time* and *location*. For the movie database, our data comes from the Stanford Movie Database [7] with information about 11500 films. The context parameters are *accompanying_people*, *mood* and *time_period*. The sizes of the datasets have two important implications for usability. First, they affect the size of the profile, i.e. the number of preferences. Then, and most importantly, they require different methods for evaluating the quality of results. While for the small dataset, we can ask users to manually provide the best results, for the large dataset, we need to use other metrics [24].

We conducted an empirical evaluation of our approach with 10 human subjects. Different users were used for each of the two datasets. For all users, it was the first time that they used the system. For each database, to ease the specification of contextual and non-contextual preferences, we created a number of default profiles based on (a) age (below 30, between 30-50, above 50), (b) sex (male or female) and (c) taste (broadly categorized as mainstream or out-of-the-beaten track). We created both contextual and non-contextual profiles. Based on the above three characteristics, one of the 12 available non-contextual profiles and one of the 12 available contextual profiles were assigned to each user. Users were allowed to modify the default profiles assigned to them by adding, deleting or updating preferences. We evaluated our approach along two lines: ease of profile specification and quality of results.

3.4.1 Profile Specification

With regards to profile specification, we count the number of modifications (insertions, deletions, updates) of preferences of the default profile that was originally assigned to the users, for both the non-contextual and contextual profiles. In addition, we report how long (in minutes), it takes users to specify/modify their profile (again, for both cases). Since for all users this was their first experience with the system, the reported time includes the time it took the user to get accustomed with the system. The results are reported in Table 3.1 for points of interest and Table 3.2 for movies. For the points of interest database, each of the default non-contextual profile has about 100 preferences, while each default contextual one nearly 650 preferences, while for the movie database, the sizes are 120 and 1 100 respectively.

The general impression is that predefined profiles save time in specifying user pref-

erences. Furthermore, having default profiles makes it easier for someone to understand the main idea behind the system, since the preferences in the profile act as examples. With regards to time, there is deviation among the time users spent on specifying profiles: some users were more meticulous than others, spending more time in adjusting the profiles assigned to them. As expected, the specification of contextual profiles is more time-consuming than the specification of non-contextual ones, since such profiles have a larger number of preferences. The size of the database also affects the complexity of profile specification basically by increasing the number of preferences and thus, the required modifications.

Table 3.1: Points-of-Interest Example: Profile Specification

	User1	User2	User3	User4	User5	User6	User7	User8	User9	User10
Non Contextual Profile										
Num of updates	15	14	8	11	15	16	19	12	10	10
Update time (mins)	13	8	5	6	6	9	16	7	9	8
Contextual Profile										
Num of updates	22	31	12	28	24	32	38	13	18	25
Update time (mins)	30	45	20	30	30	40	45	15	20	25

Table 3.2: Movie Example: Profile Specification

	User1	User2	User3	User4	User5	User6	User7	User8	User9	User10
Non Contextual Profile										
Num of updates	37	14	29	10	22	31	29	15	17	12
Update time (mins)	18	7	14	6	9	19	16	6	8	8
Contextual Profile										
Num of updates	67	49	52	91	37	72	69	41	46	37
Update time (mins)	32	28	28	55	17	44	36	22	27	46

3.4.2 Quality of Results

We compare the results of contextual queries, i.e. queries enhanced with a context descriptor, when executing them: (i) without using any of the preferences, (ii) using the non-contextual preferences and (iii) using the contextual preferences. In the case of contextual preferences, we also consider using (a) only exact matching context states, (b) only one, the most similar, context state (top-1) and (c) the three most similar context states (top-3). For similarity, we use the hierarchy distance function and to resolve ties the Jaccard distance.

Since the points of interest database has a small number of tuples, we asked the users to rank the results of each contextual query manually. Then, we compare the ranking specified by the users with what was recommended by the system, for the above five cases. For each case, we consider the best 20 results, i.e. the 20 points of interest that were ranked higher. When there are ties in the ranking, we consider all results with the

same score. We report the percentage of the results returned that belong to the results given by the user. As shown in Table 3.3, this percentage is generally high. Surprisingly, sometimes users do not conform even to their own preferences as shown by the results for exact match queries. In this case, although the context state of the preferences used was an exact match of the context state in the query, still some users ranked their results differently than the system. In such cases, traceability helps a lot, since users can track back which preferences were used to attain the results and either modify the preferences or reconsider their ranking. Note that users that customized their profile by making more modifications got more satisfactory results than those that spent less time during profile specification. Furthermore, for non exact match queries, using the most similar state (top-1) to answer a query provides only slightly better results than using the top-3 most similar states.

For the movie database, due to the large number of tuples, it was not possible for the users to manually rank all of them. Instead, users were asked to evaluate the quality of the 20 higher ranked movies in the result. For characterizing the quality of the results, users marked each of the 20 movies with 1 or 0 indicating whether they considered that the movie should belong to the best 20 ones or not, respectively. The number of 1s corresponds to the precision of the top-20 movies, or $precision(20)$, i.e. how many of the 20 movies are relevant. Furthermore, users were asked to give a specific numerical interest score between 1 and 10 to each of the 20 movies. This score is in the range [1, 5], if the previous indicator is 0 and in the range [6, 10], if the indicator is 1. We report the number of movies that were rated high (interest score ≥ 7). Finally, users were asked to provide an overall score in the range [1, 10] to indicate their degree of satisfaction of the overall result set. Table 3.4 summarizes the quality measures attained for the movie database. The detailed per user scores are depicted in Table 3.5.

Our results again, show that using contextual preferences improve quality considerably. When compared to the points of interest database, precision is lower. One reason for that is the following. In the movie example, the users were not aware of the whole dataset; they were just presented with the 20 movies in the result. Thus, they "left room" in their choices for better results that could be lying in the dataset that was not presented to them.

Table 3.3: Points-of-Interest Example: Quality of results

	User1	User2	User3	User4	User5	User6	User7	User8	User9	User10
No Preferences	10%	0%	0%	0%	0%	10%	5%	0%	0%	5%
Non-Contextual Preferences	10%	10%	0%	5%	5%	10%	15%	5%	5%	5%
Contextual Preferences										
Exact Match	100%	90%	90%	95%	90%	100%	100%	85%	100%	100%
Non Exact Match										
Top-1 state	100%	95%	90%	85%	90%	100%	100%	85%	90%	100%
Top-3 states	95%	90%	85%	95%	95%	90%	100%	75%	85%	95%

Table 3.4: Movie Example: Overall Quality of Results

	Precision(20)	**Score $\geq$ 7**	**Average Overall Score**
No Preferences	26.5%	11%	2.7
Non-Contextual Preferences	40%	19%	4
Contextual Preferences			
Exact Match	80%	71.5%	8.2
Non Exact Match			
Top-1 state	72%	58%	7.4
Top-3 states	69%	53%	7

Table 3.5: Movie Example: Quality of Results per User

		User1	**User2**	**User3**	**User4**	**User5**	**User6**	**User7**	**User8**	**User9**	**User10**
	Num of 1	5	4	7	7	4	6	4	7	3	6
No Preferences	Num of $\geq$ 7	2	0	2	3	3	2	2	4	3	1
	Overall Score	3	3	3	2	4	3	3	3	2	1
Non-Contextual	*Num of 1*	4	6	8	12	12	12	7	6	8	5
Preferences	Num of $\geq$ 7	0	1	5	4	6	7	4	3	5	3
	Overall Score	2	3	4	6	6	6	4	3	4	2
Contextual:	*Num of 1*	15	17	15	13	17	14	17	17	18	17
Exact Match	Num of $\geq$ 7	14	14	15	12	13	14	15	15	17	14
	Overall Score	8	9	8	8	7	8	8	8	9	9
Contextual:											
Non Exact Match											
	Num of 1	15	17	13	11	13	13	14	17	17	14
Top-1 state	Num of $\geq$ 7	10	14	12	9	8	12	13	12	15	11
	Overall Score	6	9	7	7	6	8	8	8	8	7
	Num of 1	13	15	12	11	13	12	14	17	17	14
Top-3 states	Num of $\geq$ 7	9	13	11	10	9	8	11	11	14	10
	Overall Score	6	8	7	7	6	6	7	8	8	7

3.5 Performance Evaluation of Context Resolution

To evaluate the performance of context resolution, we run a set of experiments using both real and synthetic profiles. As real profiles, we use the ones specified by the users in our usability study for the movie and points of interest databases. Table 3.6 summarizes the parameters used for the creation of the synthetic profiles. We consider: (a) the storage efficiency of the profile tree and the preference graph and (b) the complexity of context resolution when we use the proposed algorithms.

Table 3.6: Input Parameters for Synthetic Profiles

Parameter	Value
Number of contextual preferences	500 - 10 000
Number of context parameters	3
Data distributions	Uniform, zipf
Cardinality of context domains	50 - 1 000
Hierarchy levels	4

3.5.1 Storage

In this set of experiments, we evaluate the space (in number of cells) required to store context states using the profile tree and the preference graph as opposed to storing them sequentially (no index). With regards to the profile tree, this depends on the mapping of context parameters to the levels of the tree. Also, since the profile tree (resp. the preference graph) takes advantage of the co-occurrences of context values (resp. context states) in the profile, their size depends on the distribution of context values in the profile.

Real data sets. We built the profile tree and the preference graph for the profiles of both the movie and the points of interest datasets. In the case of the profile tree, we created profile trees for all possible mappings of context parameters to levels of the trees. For the movie database, let A stand for *accompanying_people*, M for *mood* and T for *time_period*. We call $order1$ the mapping in which A is assigned to the first level of the tree, M to the second and T to the third one, denoted as (A, M, T). Similarly, $order2$ is (A, T, M), $order3$ is (M, A, T), $order4$ is (M, T, A), $order5$ is (T, A, M) and $order6$ is (T, M, A). Accordingly, for the points of interest database, let A stand for *accompanying_people*, T for *time* and L for *location*. Then, $order1$ is order *(A, T, L)*, $order2$ is *(A, L, T)*, $order3$ is *(T, A, L)*, $order4$ is *(T, L, A)*, $order5$ is *(L, A, T)* and $order6$ is *(L, T, A)*. As shown in Figure 3.4(a), the preference graph requires 82% and 34% less space than storing preferences sequentially for the movie and the points of interest database respectively, since in the preference graph each distinct context state is stored only once. For the profile tree, the orders that result in trees with smallest sizes are, as expected, the ones that map the context parameters with large domains to levels lower in the tree ($order1$ and $order3$ for both databases). All trees occupy less space than storing preferences sequentially. The smallest profile tree representation occupies about 86% and 60% less space than a sequential representation for the movie and the points of interest database, respectively.

Synthetic data sets. We also consider the size of the proposed data structures as a function of the size of the profile, i.e. number of user preferences, and respectively, number of context states. We consider domains for the context parameters with different cardinalities, so that the mappings of context parameters to tree levels create trees with varying sizes. In particular, we use a domain with 50 values, a domain with 100 values and a domain with 1 000 values. We create profiles with 500, 1 000, 5 000 and 10 000 context states. To create context states, context values are drawn from their corresponding domain with two different distributions, uniformly or with a zipf data distribution with $a = 1.5$. As with the real profiles, we create profile trees for all 6 different mappings. Let $order1$ be the order (50, 100, 1 000) that maps the parameter whose domain has 50 values to the first level, the parameter with 100 values to the second one and the parameter with 1 000 values to the third one. Similarly, let $order2$ be (50, 1 000, 100), $order3$ (100, 50, 1 000), $order4$ (100, 1 000, 50), $order5$ (1 000, 50, 100) and $order6$ (1 000, 100, 50). The mapping of parameters with large domains lower in the tree results in smaller trees (Figures 3.4(b), (c)). For the zipf distribution (Figure 3.4(c)), the total number of cells

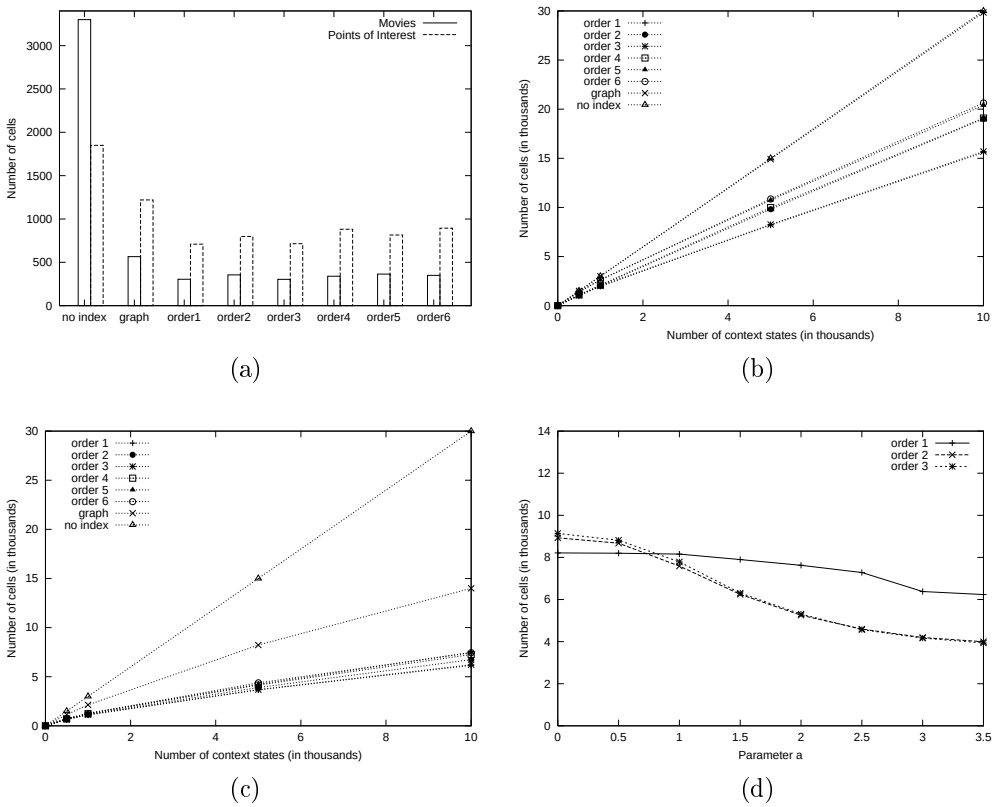


Figure 3.4: Size using (a) the real profiles and synthetic profiles with (b) uniform, (c) zipf with a=1.5 and (d) combined data distribution.

is smaller than that for the uniform distribution (Figure 3.4(b)), because "hot" values appear more frequently in preferences, i.e. more context values are the same. Concerning the preference graph, when context values are selected uniformly (Figure 3.4(b)), the size of the graph is almost equal to the size of sequential storage, since most of the produced context states are distinct. For the zipf data distribution, the graph occupies 40% less space than storing context states sequentially. Finally, the profile tree requires nearly 23% and 38% less space than the preference graph for the uniform and the zipf data distribution, respectively.

As a final note, observe that the actual size of the profile tree depends on the frequency of the values of the context parameters in the profile. Thus, even if a parameter has a large domain, if only a small number of its values appears in the profile, it should be mapped to a high level as shown next. In this experiment, the profile has 5 000 context states and the context parameters have domains with 50, 100 and 200 values. The values of the parameters with domains with 50 and 100 values are selected using a uniform data distribution and the values of the parameter with 200 values using a zipf data distribution with various values for the parameter a, varying from 0 (corresponding to the uniform distribution) to 3.5 (corresponding to a very high skew). $Order1$ is $(50, 100, 200)$, $order2$ is $(50, 200, 100)$ and $order3$ is $(200, 50, 100)$ (Figure 3.4(d)). Thus, an analysis of the actual distribution of values in the profile should be made for deciding the appropriate

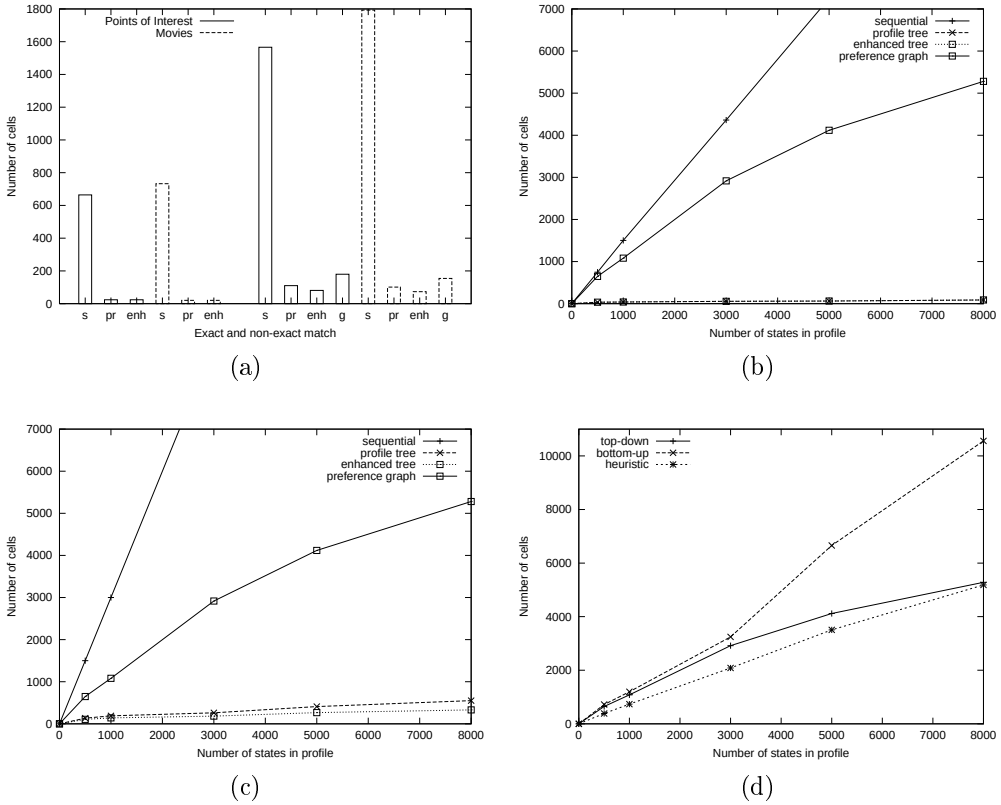


Figure 3.5: Cell accesses to find related preferences to queries using sequential scan, the profile tree, the enhanced profile tree and the preference graph for (a) the real profiles and the synthetic ones in (b) exact match and (c) non exact match and (d) for the top-down, bottom-up and the heuristic approach when the preference graph is used.

mapping, as opposed to looking only at the domain sizes.

3.5.2 Context Resolution

In this set of experiments, we study the complexity of context resolution (in term of cell accesses) using the profile tree, the preference graph and their enhancements. We use both the real profiles and synthetic profiles of different sizes. Synthetic profiles have three context parameters, each with a domain of 100 values. The values of two of them are drawn from their corresponding domain using a zipf data distribution with $a = 1.5$, while the values of the third one are selected using a uniform data distribution. This parameter is mapped to the lower level of the tree. Context parameters have hierarchies with four levels, where 75% of the context values are considered to be values at the most detailed level of the hierarchies and the rest 25% are assigned to the other three levels. The results reported are averaged over 50 randomly generated queries.

First, we compare the number of cell accesses during context resolution when (i) scanning all preferences sequentially, (ii) using the profile tree, (iii) using the enhanced profile tree and (iv) using the preference graph with a top-down traversal (Figure 3.5).

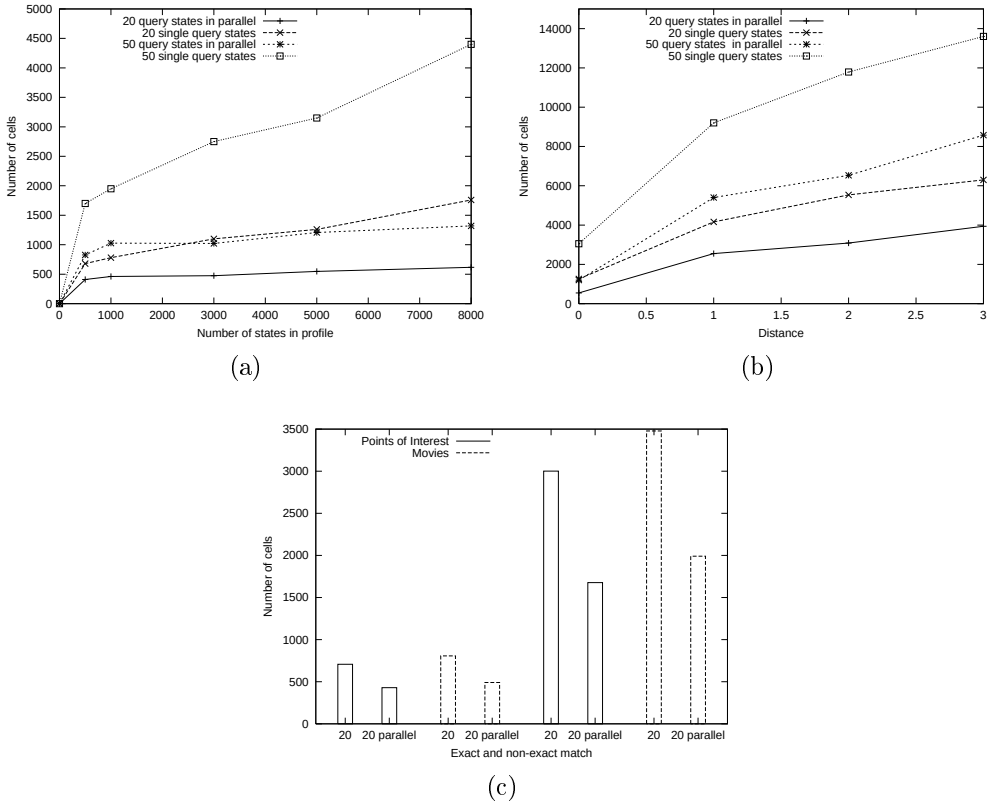


Figure 3.6: Cell accesses to find preferences related to queries using the query tree for synthetic profiles for (a) exact match and (b) non exact match and (c) for the real profiles.

The results for the real profiles are depicted in Figure 3.5(a), where with s we denote sequential scan, with pr the profile tree, with enh the enhanced profile tree and with g the preference graph, while the results of synthetic profiles are shown in Figures 3.5(b), (c). In the case of sequential scan, for exact matches, the profile is scanned until the matching state is found, while for non exact matches, we need to scan the whole profile. With the profile tree, exact match queries are resolved by a simple root-to-leaf traversal, while non exact matches need to consider multiple candidate paths. For non exact match queries, the enhanced tree reduces the number of cell accesses at about 30% with regards to the profile tree. When the preference graph is used, the number of cells accesses is the same for exact and non exact matches. Although the preference graph requires much more cells accesses than the profile tree, in turn, the preference graph requires 68% less accesses than when the preferences are sequentially scanned. Finally, in Figure 3.5(d), we count the cells accesses when the preference graph is used and the top-down, bottom-up and the heuristic approach is followed. The heuristic approach reduces the number of cells accesses at about 18% with regards to the top-down approach.

We also compare the performance (i.e. number of cell accesses) of searching for matching states for each state of the query one at a time versus matching all query states in parallel using the *QueryCR Algorithm* (Algorithm 11). We used both the real profiles and synthetic profiles with 5 000 context states. Figure 3.6(a) depicts our results for exact

match queries using the synthetic profile and query descriptors with 20 and 50 states. Searching for all states in one pass results in savings at around 40% on average. We also consider the number of cell accesses when we search not only for exact matches, but also for more general context states. Figure 3.6(b) shows the number of cell accesses taking into consideration the distance between a returned state and a searched one when we search for 20 and 50 states. Here, we save on average at around 60%. The results for the real profiles are shown in Figure 11(c). We run this experiment for exact match queries and for queries in which we search for states with distance up to 2, with query descriptors consisting of 20 context states. Using Algorithm 11, we save on average 40%.

3.6 Summary

The focus of this chapter is on annotating database preferences with contextual information. Context is modeled using a set of multidimensional context parameters. Context parameters take values from hierarchical domains, thus, allowing different levels of abstraction for the captured context data. A context state corresponds to an assignment of values to each of the context parameters from its corresponding domain. Database preferences are augmented with context descriptors that specify the context states under which a preference holds. Similarly, each query is related with a set of context states. Now, the problem is to identify those preferences whose context states as specified by their context descriptors are the most similar to that of a given query. We call this problem *context resolution*. To realize context resolution, we propose two data structures, namely the *preference graph* and the *profile tree*, that allow for a compact representation of the context-dependent preferences.

To evaluate the usefulness of our model, we have performed two usability studies. Our studies showed that annotating preferences with context improves the quality of the retrieved results considerably. The burden of having to specify contextual preferences is reasonable and can be reduced by providing users with default preferences that they can edit. We have also performed a set of experiments to evaluate the performance of context resolution using both real and synthetic datasets. The proposed data structures were shown to improve both the storage and the processing overheads. In general, the profile tree is more space-efficient than the preference graph. It also clearly outperforms the preference graph in the case of exact matches. The main advantage of the preference graph is the possibility for an incremental refinement of a context state. In particular, at each step of the resolution algorithm, we get a state that is closer to the query one. This is not possible with the profile tree.

The results presented in this chapter also appear in [113, 111].

Chapter 4

Fast Contextual Preference Scoring of Database Tuples

In this chapter, we address the problem of efficient scoring database tuples based on contextual quantitative preferences by performing preprocessing steps off-line, and in particular, by pre-computing representative tuple scores. At one extreme, we could compute all different scores for each tuple for all potential context states. Since, only a few tuples may be of interest at each context state, we propose computing scores only for relevant tuples (i.e. tuples for which there is sufficient interest). However, the number of potential scores may be still large, since the number of context states grows rapidly with the number of context parameters. For many applications, context includes a large number of parameters with domains of varying sizes. For instance, in a pervasive environment, a media player system for movies and television programs may suggest interesting programs to users according to their current context that includes their age, sex, taste as well as time, location, surrounding people, emotional state and the technical characteristics of the targeted device for playing the program. Thus, we propose computing scores only for *representative* context states. Our method for identifying context representatives exploits the hierarchical nature of context parameters and can be applied to both quantitative and qualitative preferences.

We also consider a complementary method for grouping preferences based on identifying those preferences that result in similar scores for all database tuples. This method takes advantage of the quantitative nature of preferences to group together contextual preferences that have similar predicates and scores. The method is based on a novel representation of preferences through a predicate bitmap table whose size depends on the desired precision for the resulting scoring.

In summary, in this chapter, we make the following contributions:

- We propose a suite of techniques for quickly providing users with data of interest in the case of contextual quantitative preferences.
- We consider a contextual clustering method that exploits the hierarchical nature of context parameters.
- We introduce a method for grouping those preferences that produce similar scores for all database tuples. The method is based on a bitmap representation with tunable accuracy.

Finally, we present a number of experiments on both synthetic and real data sets.

The rest of this chapter is structured as follows. In Section 4.1, we introduce the problem of scoring database tuples based on contextual quantitative preferences. Section 4.2 proposes a method for finding representative context states by exploiting the hierarchical nature of context parameters, while Section 4.3 focuses on grouping preferences by identifying those that result in similar scores. Section 4.4 discusses issues of handling the produced scores. In Section 4.5, we present our evaluation results and finally, Section 4.6 concludes the chapter with a summary of our contributions.

4.1 Contextual Preference Ranking

In this section, we present our contextual preference model and introduce the problem of scoring database tuples using contextual preferences. As a running example, we use again the movie database with schema: *Movies*(mid, title, year, director, genre, language, duration). Users express their preferences on movies. These preferences may for example depend on who is accompanying the user or the user's age or sex. For instance, cartoons may be a reasonable choice when with family, while a romantic comedy may be preferable when on a date.

4.1.1 Contextual Preference Model

To model context, we use a finite set of special-purpose attributes, called *context parameters*. Here, we distinguish between two types of context parameters: simple and composite ones. A *simple* context parameter involves a single context attribute C_i with domain $dom(C_i)$, while a *composite* context parameter C_j consists of a set of single context

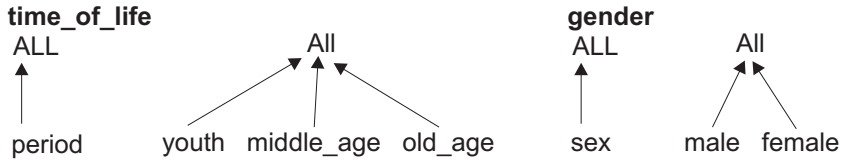


Figure 4.1: Hierarchy schema and concept hierarchy of *time_of_life* and *gender*.

attributes $C_{j_1}, C_{j_2}, \ldots, C_{j_l}$ with domains $dom(C_{j_1}), dom(C_{j_2}), \ldots, dom(C_{j_l})$ respectively and its domain, $dom(C_j)$ is equal to $dom(C_{j_1}) \times dom(C_{j_2}) \ldots \times dom(C_{j_l})$. For a given application X, we define its *context environment* CE_X as a set of n context parameters $\{C_1, C_2, \ldots, C_n\}$.

In our movie example, we consider the simple context parameters *accompanying_people*, *time_period* and *mood*. We also consider users to be part of context, so that the result of each query depends on the user submitting it. Each user is modeled by the composite context parameter *user* consisting of attributes *id*, *time_of_life* and *gender*. Thus, our context environment is the quadruple (*user, accompanying_people, time_period, mood*).

Similar to Chapter 3, context parameters take values for multidimensional domains. Figure 3.1 depicts the hierarchy schema and concept hierarchy of *accompanying_people*, *time_period* and *mood*, while Figure 4.1 depicts the hierarchy schema and concept hierarchy of *time_of_life* and *gender*.

Hierarchies support the specification of context values with various levels of detail. For a context environment with n context attributes, a *context state* cs is a n–tuple of the form (c_1, c_2, ..., c_n), where $c_i \in dom(C_i)$. For example, a context state in our example may be $((All, youth, male), friends, Th,$ *good*$)$ or $((id1, middle\_age, female), family, holidays,$ *good*$)$.

Users express their preference for sets of tuples specified using selection conditions on some of the attributes of the tuples by rating them using a numerical score. The score is a real number between 0 and 1 which expresses their degree of interest for the specified tuples. Value 1 indicates extreme interest, while value 0 indicates no interest. Preferences are annotated with context information to denote the context state under which the preference holds. As in Chapter 3, the meaning of a contextual preference $p =$ (cs, $Pred$, *score*) is that in context state cs, all database tuples t for which $Pred$ holds are assigned the indicated score. For instance, a preference ((*id1, youth, male*), *friends, holidays, good*), (*genre = comedy*), 0.9) denotes that user with id1 who is a young male, when accompanied with friends during holidays and in good mood enjoys seeing movies of genre comedy.

We recall that it is not necessary for a preference to depend on all context attributes. This can be expressed by assigning the value *All* to the corresponding context attribute. For instance, the preference (((*All, youth, All*)), *All, holidays, All*), (*genre = comedy*), 0.9) means that all young people like to see comedies during holidays independently of the values of the other context parameters. For simplicity, in following, we shall skip *All* values in the context part of the preference and for example, simply use ((*youth, holidays*),

$(genre = comedy)$, 0.9) to express the preference above.

We call the set of contextual preferences that hold for an application, *profile* P. By $Context(P)$, we denote the set of context states cs that appear in at least one preference in P. We assume that such profiles are available. In practice, preferences may be, for example, given by users explicitly or may be deduced by say the previous behavior of the same or similar users. A practical way to create P, considered in [113], is by assembling a number of default profiles and allowing users to update them appropriately.

4.1.2 Problem Formulation

Each query submitted by a user is associated with one or more context states. Typically, the context implicitly associated with a query corresponds to the current context, that is, the context surrounding the user at query submission time. To capture the current context, many context-aware applications use various devices for determining the values of the relevant context parameters, such as temperature sensors or GPS-enabled devices for location-related attributes. Besides such implicit context augmentation, queries may be explicitly enhanced with context states for example for posing exploratory queries such as what is a good movie to watch with my *family* this coming *weekend*.

Now given the query context states $Context(q)$ of a query q, we would like

(1) to identify the set $P_q \subseteq P$ of preferences $(cs, Pred, score)$ for which $cs = cs_q$, for some $cs_q \in Context(q)$ and then,

(2) use them to compute a score for each tuple t in the result of q.

The first problem is complicated by the fact that for some cs_q in $Context(q)$, there may be no preference $(cs, Pred, score)$ in the profile P, with $cs = cs_q$, that is, $cs_q \notin Context(q)$. Note, that the set of all possible context states for a context environment with n parameters is equal to $|dom(C_1)| \times |dom(C_2)| \times \ldots |dom(C_n)|$. In practice, the profile contains preferences only for a small number of such states. To address this, we use those preferences in P that have the most similar context states. That is, for each query context state cs_q, we use the preferences $(cs, Pred, score)$ in P with $min_{cs \in Context(P)} dist_S(cs, cs_q)$. We defer the definition of distance $dist_S$ between context states to Section 4.2. The score of a tuple with regards to a set of context states is defined based on Definition 3.15.

Now, the second problem can be expressed as follows:

Problem Definition. Assume a database instance r, a profile P and a query q with a set of context states $Context(q)$. Let $CS \subseteq Context(P)$ be the set of context states cs with the minimum $dist_S(cs, cs_q)$, $cs_q \in Context(q)$, that is, the context states that are the most similar to the context states of the query. The *contextual scoring problem* is to rank all tuples t in the result of q based on the aggregate score $score(t, CS)$.

For computing the scores of all tuples in the result set, a solution that involves no pre-computation is to first find the set of context states $Context(q)$, compute the scores

of all tuples t in the result and then rank them based on these scores. Performance can be improved by performing preprocessing steps off-line.

One approach would be to compute the scores of each tuple for each potential context state. Assuming a large database and that only a few tuples are of interest at any given context, computing a score for all database tuples for each context state will result in both wasting resources and slow query responses. Since, the number of possible context states grows rapidly with the number of context attributes, we could instead compute the scores for all states that appear in the profile and then combine the scores of the most similar ones on-line.

Since the number of context states that appear in a profile can still be large, we propose two approaches for finding representative scores to pre-compute. The first approach constructs clusters of preferences, considering as similar those preferences that have either the same or similar context states. The second one clusters preferences that lead to similar scores for database tuples.

After constructing the clusters of preferences, we compute for each cluster, an interest score for each database tuple using the preferences of this cluster. Furthermore, instead of storing scores for all database tuples for each cluster, we just store the nonzero ones. Then, for each query, we can search for the most similar to the query cluster and quickly provide the best results, that is, the results with the largest scores.

In a nutshell, our solution framework for addressing the above problem consists of the following components:

1. Having defined preferences that hold under different circumstances, we cluster them according either to
   (a) their context part, thus creating clusters of preferences applicable to similar context states, or
   (b) their non-context part, i.e. the predicate and score part, thus creating clusters of preferences that produce similar scores.
2. Using the preferences of each cluster, we compute an interest score for each tuple for the given cluster.
3. For a submitted query, we search for the most similar to the context of the query clusters. Using the scores of tuples of the returned clusters, we quickly rank the results based on the computed scores.

In the following two sections, we describe how we cluster preferences for the case of context state similarity and the case of predicate similarity.

4.2 Finding Representative Context States

Instead of computing aggregate scores for all tuples for all potential context states, we identify representative context states and pre-compute scores according to them. Com-

puting interest scores using only representative context states is based on the assumption that preferences defined for similar context states would result in producing similar scores for most tuples.

In the following, we first define the notion of similarity or, equivalently, distance between context states. Then, we use a simple clustering algorithm that groups similar context states and selects one context state per cluster as a representative state.

4.2.1 Similarity between Context States

Defining similarity between context states is a difficult problem, since context similarity is in general application dependent. Here, we take a rather generic, syntactic approach that exploits the hierarchical domains of each context parameter. First, we define similarity for each of the context parameters.

A direct method to compute the distance between two values of a context parameter is by relating their distance with the length of the minimum path that connects them in their associated hierarchy. However, this method may not be accurate, when applied to attributes with large domains and many hierarchy levels. This is because values in upper levels of the hierarchy are intuitively less similar than values in lower levels connected with paths of the same length. For instance, in our simple example of the *time_period* hierarchy, when considering only the path length, values Tu and W have the same distance with each other as value *working_days* has with $Weekend$. Moreover, the distance between Tu and W is the same as Tu and All, whereas, Tu is intuitively more similar to W than to All.

Following related research on defining semantic similarity between terms (e.g. [86]), in defining the distance between two values of a context parameter, we take into account both their path distance and the depth of the hierarchy levels that the two values belong to. Let $lca(c_1, c_2)$ be the lowest common ancestor of context values c_1 and c_2. The path and depth distance between two values are defined as follows.

Definition 4.1 (Path distance)**.** The path distance $dist_P(c_1, c_2)$ between two context values $c_1 \in dom_{L_j}(C_i)$ and $c_2 \in dom_{L_k}(C_i)$:

- is equal to 0, if $c_1 = c_2$,
- is equal to 1, if c_1, c_2 are values of the lowest hierarchy level and $lca(c_1, c_2)$ is the root value of their corresponding hierarchy, or
- is computed through the f_p function $(1 - e^{-\alpha \times \rho})$, where $\alpha > 0$ is a constant and ρ is the minimum path length connecting them in the associated hierarchy.

The f_p function is a monotonically increasing function that increases as the path length becomes larger. The above definition of *path distance* ensures also that the distance is normalized in $[0, 1]$.

Definition 4.2 (Depth distance)**.** The depth distance $dist_D(c_1, c_2)$ between two context values $c_1 \in dom_{L_j}(C_i)$ and $c_2 \in dom_{L_k}(C_i)$:

- is equal to 0, if $c_1 = c_2$,
- is equal to 1, if $lca(c_1, c_2)$ is the root value of their corresponding hierarchy, or
- is computed through the f_d function $(1 - e^{-\beta/\gamma})$, where $\beta > 0$ is a constant and γ is the minimum path length between the $lca(c_1, c_2)$ value and the root value of the corresponding hierarchy.

The f_d function is a monotonically increasing function of the depth of the lowest common ancestor. Again, the definition of *depth distance* ensures distances within the range $[0, 1]$. Having defined the path and the depth distances between two context values, we define next their overall distance.

Definition 4.3 (Value distance)**.** The value distance between two context values c_1 and c_2 is computed as:

$$dist_V(c_1, c_2) = dist_P(c_1, c_2) \times dist_D(c_1, c_2).$$

For example, the path distance between values *summer* and *working_days* is $1 - e^{-3} \simeq 0.95$, their depth distance is 1, and so, their value distance is $1 \times 0.95 = 0.95$. Whereas values *holidays* and *summer* have value distance equal to $(1 - e^{-1\times1}) \times (1 - e^{-1/1}) \simeq 0.39$. This means that the value *summer* is more closely related to *holidays* than to *working_days* as expected. In both examples, we assume that $\alpha = \beta = 1$.

Note that to compute the value distance $dist_V$, we use the independent $dist_P$ and $dist_D$ distances. This independence enables us to combine them in different ways by giving different weights of interest. To do this, we may assign different values to the constants α, β. In particular, for constant values greater than 1, the corresponding distance increases, while values within the range (0, 1) result in smaller distances.

Having defined the distance between two context values, we can now define the distance between two context states. To achieve this, we use a simple weighted sum, but other methods of aggregation are also possible.

Definition 4.4 (State distance)**.** Given two context states $cs^1 = (c_1^1, c_2^1, \ldots, c_n^1)$ and $cs^2 = (c_1^2, c_2^2, \ldots, c_n^2)$, the state distance is defined as:

$$dist_S(cs^1, cs^2) = \sum_{i=1}^{n} w_i \times dist_V(c_i^1, c_i^2),$$

where each w_i is a context parameter specific weight.

The above weights are normalized, such that, $\sum_{i=1}^{n} w_i = 1$. The weight assigned to each context parameter is application dependent, since for some applications, some context parameters may be more influential than others. Again, in this work, we take a generic approach and assign weights to each context parameter according to the cardinality of its domain. In particular, we assign larger weights to parameters with smaller domains, considering a higher degree of similarity among values that belong to a large domain.

It is easy to show that the distance relationship between context states is reflexive ($dist_S(cs_1, cs_1) = 0$), and symmetric ($dist_S(cs_1, cs_2) = dist_S(cs_2, cs_1)$). However, it does not satisfy the triangle inequality ($dist_S(cs_1, cs_2) \leq dist_S(cs_1,\ cs_3) + dist_S(cs_3, cs_2)$), because of the semantic way of defining distances among context values, as the following example shows. Assume three context states cs_1, cs_2, cs_3 with a single context parameter, say *time_period*, and in particular, $cs_1 = (Sunday)$, $cs_2 = (summer)$, $cs_3 = (All)$. Assuming further that α, β are equal to 1, $dist_S(cs_1, cs_2) \leq dist_S(cs_1, cs_3) + dist_S(cs_3, cs_2)$ does not hold, because $1 \leq (1 - e^{-2}) + (1 - e^{-2})$ does not hold.

4.2.2 Contextual Clustering

To group preferences with similar context states, we use a typical hierarchical agglomerative clustering method that follows a bottom-up strategy. Initially, the *d-max* algorithm (Algorithm 12) places each context state in a cluster of its own. Then, at each step, it merges the two clusters with the smallest distance. The distance between two clusters is defined as the maximum distance between any two context states that belong to these clusters. The algorithm terminates when the closest two clusters, i.e. the clusters with the minimum distance, have distance greater than d_{cl}, where d_{cl} is an input parameter. Finally, for each produced cluster, we select as representative context state, the state in the cluster that has the smallest total distance from all the states in its cluster. Formally:

Definition 4.5 (Representative context state)**.** Let cl_i be a cluster produced by the d-max algorithm that consists of a set CS_{cl_i} of m context states, cs_{i_j}. The representative of cl_i is the context state $cs \in CS_{cl_i}$, with the minimum $\sum_{j=1}^{m} dist_S(cs, cs_{i_j})$ value.

Using the *d-max* algorithm, any two context states cs_1, cs_2 that belong to the same cluster have distance $dist_S(cs_1, cs_2) \leq d_{cl}$. Therefore, the following property holds.

Property 4.1. *Given a cluster distance d_{cl}, each cluster cl produced by the* d-max *algorithm has context states, such that, any pair of context states cs_1, $cs_2 \in cl$ have distance $dist_S(cs_1, cs_2) \leq d_{cl}$.*

Proof. From step 2, of the *d-max* algorithm, we merge the closest two clusters, if their distance is less or equal to d_{cl}. This distance represents the maximum distance between a context state cs_1 of the first cluster and a context state cs_2 of the second one. Therefore, any two context states cs_1, cs_2 that belong to the same cluster, have distance $dist_S(cs_1, cs_2)$ $\leq d_{cl}$. ∎

After generating the clusters of preferences, we compute for each of them an aggregate score for each tuple specified in any of its preferences (using the definition of the aggregate tuple score). For each produced cluster cl_i, we maintain a table, called *scoring table*, $cl_i Scores$(tuple_id, score), in which we store in decreasing order only the scores of tuples that satisfy at least one of the predicates in the preferences of the cluster. That is, we do not maintain scores for all tuples, but only for those having nonzero scores. Each time

d-max Algorithm

Input: A set of preferences with context states cs_i, a distance value d_{cl}.

Output: A set of clusters.

Begin

1. Create a cluster for each context state cs_i.
2. Repeat.
   2.1 If the minimum distance among any pair of clusters
   is smaller than d_{cl}.
      2.1.1 Merge these two clusters.
   2.2 Else, end loop.
3. Compute the representative context state of each produced cluster.

End

Algorithm 12: *d-max* Algorithm

a query is submitted, we search for the most similar cluster or clusters, that means, for the clusters whose representative context state is the most similar to the query context. Then, using the scoring table of the corresponding clusters, we can quickly retrieve the tuples with the highest scores.

It is straightforward (by Definition 3) that:

Property 4.2. *Let cs be a context state and CS a set of context states. If $cs \in CS$, then for any $t \in r$, $score(t, CS) \geq score(t, cs)$.*

This means that the score of a tuple computed using the representative context state is no less than the score of the tuple computed using any of the context states belonging to the cluster. In other words, if the context state that is the most similar to the query context belongs to the cluster whose representative context state is the most similar to the query, then the score that our approximation approach computes for a tuple cannot be lower than the exact one. That is, we may overrate a tuple, but we never underrate it.

4.3 Predicate Clustering

Context-based clustering groups together similar context states. In this section, we consider an alternative approach for clustering context states that aims at grouping together context states that produce similar scores for most database tuples. To this end, we introduce a bitmap representation for the preferences applicable to a context state cs.

Let $\mathcal{P}$ be the set of all predicates that appear in P and l be the number of all distinct scores. We define the preference matrix $B(cs)$ for a context state cs as an $l \times |\mathcal{P}|$ two-dimensional array, where $B(cs)[i, j] = 1$, if and only if, there is a preference that holds under context cs and gives to tuples for which predicate j holds an interest score equal to i. In particular:

Definition 4.6 (Preference matrix)**.** A preference matrix $B(cs)$ for a context state cs is a bitmap $l \times |\mathcal{P}|$ array, where $|\mathcal{P}|$ is the number of all distinct predicates and l the number of all distinct scores in P, such that, $B(cs)[i,j] = 1$, if and only if, there is a preference $(cs, j, i) \in P$.

Clearly, if the matrices $B(cs_1)$ and $B(cs_2)$ of two context states cs_1 and cs_2 are the same, then all database tuples have the same scores for the context states cs_1 and cs_2. Since these preference matrices can be very large, we define approximations of them as follows.

Definition 4.7 (Predicate representation)**.** A predicate representation of a context state cs and score s, $BV(cs, s)$, $0 \leq s \leq 1$, is a binary vector of size $|\mathcal{P}|$, such that, $BV(cs, s)[j] = BOR_{i \geq s} B(cs)[i, j]$, where BOR is the binary OR operation.

This means that if $B(cs, s)[j] = 1$, then the score of every tuple t for which predicate j is true has score in context state cs at least equal to s, that is:

Property 4.3. *Let $BV(cs, s)$ be the predicate representation for context state cs and score s. If $BV(cs, s)[j] = 1 \Rightarrow \forall\ t \in r$, for which predicate j holds, $score(t, cs) \geq s$.*

Proof. Assume that $BV(cs, s)[j] = 1$. From Definition 4.7, this means that, for some i, $i \geq s$, $B(cs)[i, j] = 1$. Thus, from Definition 4.6, a preference (cs, j, i) belongs to P, thus from the definition of contextual preferences, for each tuple t for which preference j holds, $score(t, cs) \geq i$, which proves the property. ∎

Based on this property, the following properties relate the predicate representations for two context states cs_1 and cs_2 with the interest scores they assign to tuples.

Property 4.4. *Let $BV(cs_1, s)$ and $BV(cs_2, s)$ be the predicate representations of two context states cs_1 and cs_2 for score s.*

(a) If $\forall\ j$, $BV(cs_1, s)[j] = 1 \Rightarrow BV(cs_2, s)[j] = 1$, then the set of tuples that have score larger than s in cs_2 is a superset of the set of tuples that have score larger than s in cs_1.

(b) If $\forall\ j$, $BV(cs_1, s)[j] = 1 \Leftrightarrow BV(cs_2, s)[j] = 1$, then the set of tuples with scores larger or equal to s are the same in both context states.

Proof. Proof of (a): Let t be a tuple that has score larger than s in cs_1, $score(t, cs_1) \geq s$. This means that there is at least one preference (cs_1, j, s') with $s' \geq s$ that belongs to profile P, for which predicate j holds in t. From Definition 4.7, this means that for this j, $BV(cs_1, s)[j] = 1$. Thus, $BV(cs_2, s)[j] = 1$, and from Property 4.3, it holds $score(t, cs_2) \geq s$.
Proof of (b): This holds trivially from (a). ∎

Table 4.1: Overall predicate representation matrix BM for $friends$

	$horror$	$Hitscock$	$Spielberg$
0.8	1	0	0
0.7	1	1	0
0.6	1	1	0

The distance between two binary vectors depends on the number of bits they differ at. In particular, let V_1 and V_2 be two binary vectors of size m, then $diff = \sum_{i=1}^{m} |V_1(i) - V_2(i)|$. For computing the distance between two binary vectors, we shall use the well known Jaccard coefficient that ignores the negative matches, that is, the bits for which both vectors have values equal to 0. Let pos be the number of bits that are equal to 1 for both V_1 and V_2.

Definition 4.8 (Vector distance)**.** The distance of two vectors V_1 and V_2 of size m is equal to: $dist_V(V_1, V_2) = \frac{diff}{diff+pos}$ if $diff + pos \neq 0$ and 1 otherwise.

It is clear that given two context states, the number of bits that their predicate representations differ at is an indication of the number of tuples that they rank differently. Specifically, from Property 4.4(b), if for two context states cs_1 and cs_2, $dist_V(BV(cs_1, s), BV(cs_2, s)) = 0$, then the set of tuples with score greater or equal to s associated with each BV is the same.

Note that some predicates may hold for more tuples than others. If such information regarding the selectivity of the predicates is available or can be estimated, then it is possible to consider a weighted version of $diff$ as follows: $\sum_{i=1}^{m} w(i) |V_1(i) - V_2(i)|$, where each $w(i)$ is set to be proportional to the selectivity of the predicate i.

Now, instead of storing bitmap representation vectors BV for all distinct interest scores s_i, we create an overall bitmap representation matrix BM with only b rows, one for each score $s_1, s_2, \ldots, s_b$, with $0 \leq s_1 < s_2 < \ldots < s_b \leq 1$. In particular:

Definition 4.9 (Overall predicate representation)**.** An overall predicate representation matrix BM for a context state cs and b scores $s_1, s_2, \ldots, s_b$, with $0 \leq s_1 < s_2 < \ldots < s_b \leq 1$ is a bitmap $b \times |\mathcal{P}|$ array, where $|\mathcal{P}|$ is the number of predicates in P, such that, $BM(cs)[i, j] = BV(cs, s_i)[j]$, $1 \leq i \leq b$, $1 \leq j \leq |\mathcal{P}|$.

Simple overall predicate representation matrices with $b = 3$ for the preferences: p_1 = *(friends, genre = horror, 0.8)*, p_2 = *(friends, director = Hitscock, 0.7)*, p_3 = *(alone, genre = horror, 0.7)* and p_4 = *(alone, director = Spielberg, 0.6)*, are depicted in Tables 4.1 and 4.2.

Next, we define the distance between two such predicate representation matrices:

Definition 4.10 (Overall representation distance)**.** The distance between two overall predicate representation $b \times |\mathcal{P}|$ matrices $BM(cs_1)$ and $BM(cs_2)$ of two context states cs_1 and cs_2, is defined as: $dist_{BM}(BM(cs_1), BM(cs_2)) = \frac{\sum_{i=1}^{b} dist_V(BV(cs_1,s_i), BV(cs_2,s_i))}{b}$.

Table 4.2: Overall predicate representation matrix BM for *alone*

	horror	*Hitscock*	*Spielberg*
0.8	0	0	0
0.7	1	0	0
0.6	1	0	1

For instance, the distance between the matrices that are shown in Tables 4.1 and 4.2 is equal to: $\frac{1+1/2+2/3}{3} = \frac{13}{18}$. Note that the distance between overall representation matrices takes values within the range $[0, 1]$.

For the distance among any predicate representation matrices BM_1, BM_2 and BM_3, the following properties hold:

1. $dist_{BM}(BM_1, BM_1) = 0$ (reflexivity);
2. $dist_{BM}(BM_1, BM_2) = dist_{BM}(BM_2, BM_1)$ (symmetry);
3. $dist_{BM}(BM_1, BM_2) \leq dist_{BM}(BM_1, BM_3) + dist_{BM}(BM_3, BM_1)$ (triangle inequality).

Thus, the overall representation distance is a metric.

We could also consider weighted versions where rows that correspond to higher scores influence the overall distance more than rows that correspond to smaller scores. Also, note that these matrices can still be very large, when the number of predicates is large. One can consider reducing the number of columns by grouping together similar predicates or by ignoring predicates with small selectivity. We leave this issue as future work.

Using distances among overall predicate matrices, we create clusters of preferences that result in similar scorings of database tuples. To do this, we use the *d-max* algorithm. Again, initially, each preference with a specific context state is placed in its own cluster. At each step, we merge the two clusters with the smallest distance, computing the distance as in Definition 4.10. The distance between two clusters is defined as the maximum distance between any two overall predicate representation matrices of context states that belong to these clusters. The algorithm terminates when the closest two clusters have distance greater or equal to s_{cl}, where s_{cl} is an input parameter.

Observe that we use the predicate matrices only to group similar preferences and not for computing scores. After the clusters have been created, we compute for each cluster an aggregate score for the tuples. As in our contextual clustering method, for each cluster cl_i, we maintain a scoring table $cl_i Scores$(tuple_id, score), where we store in decreasing order only the scores of tuples that satisfy at least one of the predicates in the preferences of the cluster (and not the scores equal to 0). When a query is submitted, we search for the most relevant cluster or clusters, that means, for the clusters that contain the

preference(s) with the same or the most similar context state (or states in the case of ties) to the query context state.

From the way an aggregate tuple score is computed within a cluster, Property 4.2 holds. This means that, when using the predicate clustering approach, the score of a tuple is no less than the score computed using any preference of the cluster. Therefore, as with contextual clustering, we may overrate a tuple, but we never underrate it.

4.4 Other Issues

Since there is potentially one different score for each database tuple per context state, the number of these scores and thus, database rankings can be very large. So far, we have addressed the problem of reducing the number of precomputed database scores through clustering. Next, we discuss further the issue of handling the interesting rankings after they have been identified through our clustering algorithms. We also consider how to maintain the rankings in the presence of profile and database updates.

4.4.1 Online Phase

Once the interesting clusters and thus rankings are identified, there are many alternative ways to materialize them. Since, our focus is on determining the interesting rankings, rather than on their efficient realization, we have adopted the following simple approach. We assume that the produced scores for each interesting cluster are stored in special tables, called *scoring tables*, with two attributes the *tuple_id* and the associated score. There is one scoring table per interesting ranking, that is, per cluster. The scoring tables are sorted by score.

When a contextual query q is submitted, the scoring table that is associated with its context cs_q is used. In the case of contextual clustering, this is the table that corresponds to the cluster whose representative context state cs is the most similar to cs_q. In the case of predicate clustering, we use the table corresponding to the cluster that contains either cs_q or if cs_q does not appear in the profile, the context state that is the most similar to cs_q.

Locating the appropriate scoring table can be achieved by maintaining an additional directory table $(C_1, C_2, \ldots, C_n, \textit{table\_id})$, where C_i, $1 \leq i \leq n$, is a context attribute and *table_id* is the scoring table associated with the respective context state. The selection of the appropriate table can be made more efficient by deploying indexes on the context attributes that appear in the profile P. Such a prefix-based data structure, termed *profile tree*, was introduced in Chapter 3.

Moreover, the physical storage of the precomputed results can be improved. For instance, we can avoid computing the scores for each tuple and storing them in the scoring table. Instead, we could simply cluster preferences in the profile P and build appropriate indexes on the database tuples based on the predicates that appear in the preferences

of each cluster. For example, for each cluster, we can just index the tuples that satisfy predicates associated with high scores. In this case, again we first locate the appropriate cluster based on the context query cs_q. Then, we use the associated predicate indexes to find the tuples with the highest scores.

When more than one cluster are used to compute the results of a query, we can use a $top-k$ algorithm (such as, FA, TA or their variations [47, 49, 52, 97]) to combine the ordered lists maintained in the scoring tables $cl_i Scores$ of the related clusters.

Furthermore, we point out that in this paper, as in many search engines and similar to [10], we rank tuples independently of the specific query. Ranking the results of ad-hoc SQL queries in a context state cs_q can be achieved by joining their results with the scoring table applicable to the specific context state.

As a final note, consider that the two clustering approaches can be applied together. For example, we can apply predicate clustering first. Then, we can apply contextual clustering to group the clusters produced based on the similarity of their context states.

4.4.2 Handling Updates

Pre-computing results increases the efficiency of queries but introduces the overhead of maintaining the results in the presence of updates. In this section, we discuss handling insertions and deletions of contextual preferences and database tuples. An update is considered as a delete followed by an insert.

When a database tuple is added (deleted), we just need to add (delete) its entries in all scoring functions. Clustering is not affected.

In the case of profile updates, let us, first, consider the case of adding or deleting a preference for a context state cs that already exists, that is, for a context state for which other preferences are already in the profile. In the case of contextual clustering, the clustering itself is not affected, since it is solely based on the context states. We just need to update the scores in the scoring table of the cs cluster of all tuples affected. This may be expensive, since in the absence of indexes, this may require scanning the whole database. On the other hand, in the case of predicate clustering, adding a preference for an existing context state cs may affect clustering. This happens when the addition of the preference causes the distance of the predicate table for cs to exceed the threshold distance s_{cl} from the other tables in its cluster. This means that cs must be moved to another cluster. Again, we need to update the scores in the associated scoring table of the previous and the new cluster of cs. The same holds for the deletion of a preference.

Let us now consider the addition of preferences for a new context state, ncs. In the case of contextual clustering, this requires finding an appropriate cluster for ncs and updating the associated scoring table. Analogously, in the case of predicate clustering, the predicate table for ncs is computed and ncs enters the appropriate cluster based on its predicate table. The scoring table of the cluster that received ncs must be updated in both cases.

The above operations may be expensive. However, typically, updates, and especially

profile updates, are not as frequent as queries. Furthermore, one can consider batch variations, where updates are not applied immediately but say periodically or when their number exceeds some threshold. In between, the users get results that may be less accurate. In such cases, various optimizations are possible by aggregating the effects of a number of updates and applying them collectively.

4.5 Evaluation

Contextual clustering is based on the premise that preferences for similar context states produce similar scores. We first run a related experiment to explore this. Then, we evaluate both contextual and predicate clustering regarding the number of representative rankings and the associated accuracy using both real and synthetic datasets.

4.5.1 Context and Preference Similarity

The goal of this experiment is to show that often preferences for similar context states are also similar. Since there are no real large profile data sets defined using our contextual preferences, we used a real dataset of movie ratings that includes 1000 users, 4000 movies and 150000 ratings [2]. Ratings are of the form (*user_id, movie_id, rating_value*), with *rating_value* in the range $[1, 5]$. For users, there is information available of the form (*user_id, sex, age, occupation*) that we use as our context environment. We constructed simple predicates that involve the genre of the movies by averaging the rates assigned by each user to movies of each genre. We consider five values for the genre attribute namely, comedy, action, thriller, horror and drama. Using the profile such constructed, we show how preferences vary with context (i.e. user attributes). We compute the distance between two context states (users) using the distance between two context states (Definition 4.4) with equal weights assigned to each of the four parameters user_id, sex, age, and occupation. We compute the distance between two ratings using the overall representation distance (Definition 4.10). The ratings of each user are represented with an overall predicate representation 5×5 matrix, where there is one row for each rating (1 to 5) and one column for each movie genre. As shown in Figure 4.2, the distance between ratings increases as the distance between users increases.

4.5.2 Contextual and Predicate Clustering

We run a set of experiments using both synthetic and real data sets to evaluate the contextual and the predicate clustering approaches. In both cases, we use a variation of the *d-max* clustering algorithm that uses as input the number of clusters instead of the distance. This allows us to directly relate the number of clusters with the quality of the rankings.

Concerning the synthetic data sets, we use a database with 100000 tuples. The

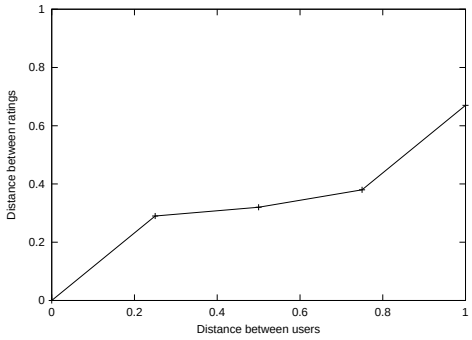


Figure 4.2: Distance of rankings as a function of distance between users.

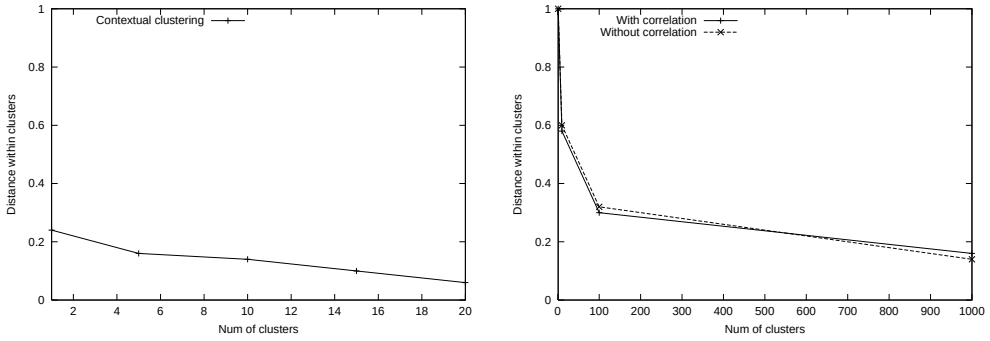
database schema consists of a single relation with 5 attributes. A synthetic profile consists of 10000 contextual preferences, each involving 3 context attributes and 2 database attributes. Context and non-context (attribute) values are selected using a zipf data distribution with a = 1.5 from context domains with 100 values and 4 hierarchy levels, and respectively, from domains with 50 values. We consider two cases for producing synthetic profiles. In the first case, there is no correlation between the context values and the other part of the preferences. In the second case, we construct correlated profiles, that is, we produce preferences for which similar context states have similar predicates and scores.

Regarding the real data sets, we use a real database with information about movies from the Internet Movies Database (IMDB) [1]. In particular, we extract from IMDB movies with language English, French, Greek, German, Spanish or Japanese. Our subset consists of nearly 40000 movies. The database schema consists of a single relation: Movies(mid, title, year, director, genre, language, duration). We run our prototype implementation for 10 users. Each user was asked to express contextual preferences for movies. To express such contextual preferences, users used the context parameters that are depicted in Figure 3.1 (namely, accompanying_people, mood and time_period) and 1 attribute of the movies relation. Each user provided about 100 preferences. We use these preferences to construct a real profile having nearly 1000 preferences.

We count the average distance within the produced clusters using the *d-max* clustering algorithm for different number of produced clusters (*Experiment I*). This is an indication of the similarity of the preferences that belong to the same cluster. Then, we evaluate the quality of the returned results for a query (*Experiment II*).

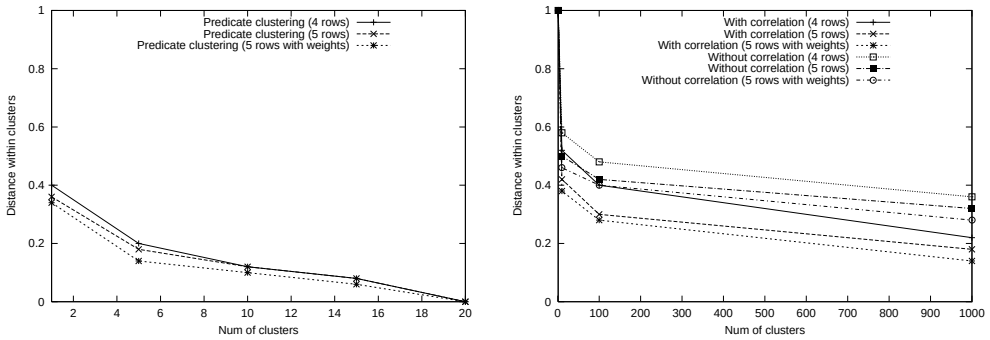
Experiment I. In this set of experiments, we vary the maximum number of clusters (i.e. rankings) and report the average distance between context states within each cluster for the contextual clustering approach and the average overall representation distance within each cluster for the predicate clustering approach.

Figure 4.3 reports the average distance among context states within the produced clusters for the contextual clustering approach for the real preferences (Figure 4.3a) and for the synthetic ones (correlated and non-correlated case) (Figure 4.3b). As expected, the correlation between the contextual and the non contextual part of a preference does



(a) (b)

Figure 4.3: Distance between context states within the produced clusters for the contextual clustering approach, for (a) real and (b) synthetic data sets.

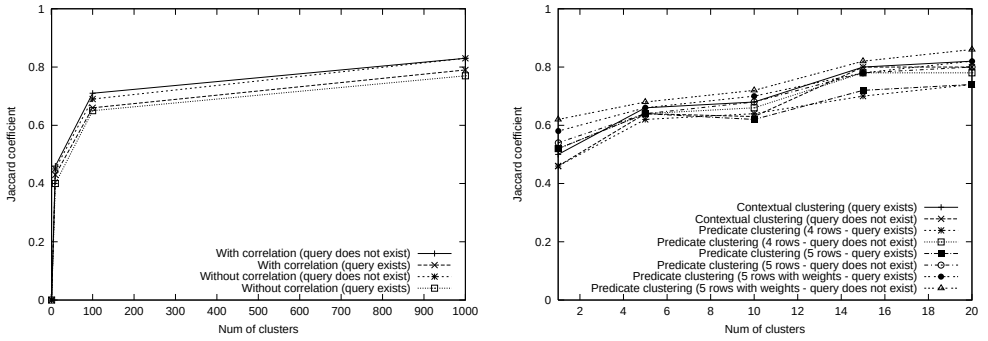


(a) (b)

Figure 4.4: Distance between context states within the produced clusters for the predicate clustering approach, for (a) real and (b) synthetic data sets.

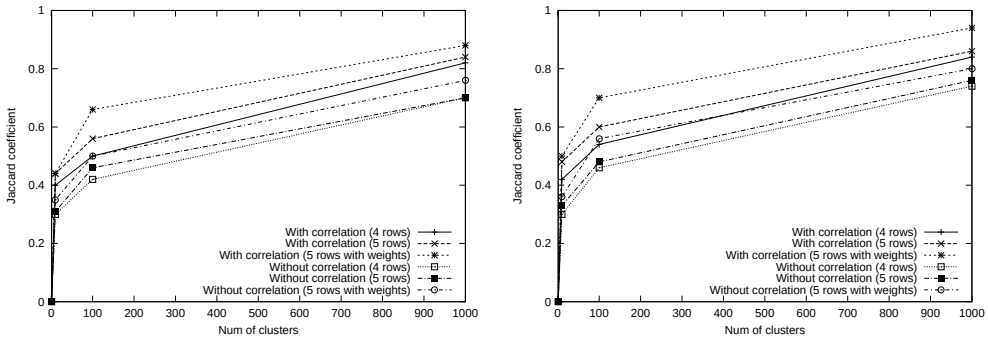
not affect the distance between the context states, since contextual clustering just uses the context part. For both the synthetic and the real data sets, the distance decreases with the number of clusters. Note that when using real preferences, the number of rankings (respectively clusters) is small because of the small cardinalities of context domains and the high degree of similarity among user preferences.

Figure 4.4 depicts results for the predicate clustering approach for different values of b, where b is the number of rows (scores) of the predicate matrix for the same real (Figure 4.4a) and synthetic (Figure 4.4b) profiles. We use matrices with 4 and 5 rows. In addition, for the case of a 5 row matrix, we consider a weighted version for computing the similarity between two matrices, where the 2 rows that refer to the two highest scores are assigned larger weights. Again, we report the distance among context states within the produced clusters for different numbers of clusters. In the case of predicate clustering, correlation reduces the distance among context states within a cluster at around 10%. The above distance is reduced further by using more accurate matrices, that is, matrices with more rows. The weighted version achieves an additional reduction of around 5%.



(a) (b)

Figure 4.5: Result quality for different number of produced clusters (a) for synthetic data sets for the contextual clustering approach and (b) for real data sets for both approaches.



(a) (b)

Figure 4.6: Result quality for different number of produced clusters, for the predicate clustering approach when (a) query states exist in the profile or (b) do not exist.

Experiment II. In this set of experiments, we evaluate the quality of results. In particular, assume that *Results(d-max)* is the set of the top-k tuples (that is, the k tuples having the largest scores) computed using the *d-max* algorithm and $Results(opt)$ is the set of top-k tuples computed using the contextual preferences that are most similar to the query without pre-computation. We compare these two sets using the Jaccard coefficient defined as:

$$\frac{|Results(d-max) \cap Results(opt)|}{|Results(d-max) \cup Results(opt)|}.$$

The Jaccard coefficient takes values between 0 and 1 and the higher its value, the more similar the two top-k tuple sets. We report the results for $k = 20$. When there are ties in the ranking, we consider all results with the same score.

For all cases, we consider two kinds of queries: queries whose context state is included in the profile and queries whose context state is not in the profile, thus, a similar one is used. In particular, Figure 4.5a depicts the results of the contextual clustering approach (with and without correlation). When the query states do not exist in the profile, the Jaccard coefficient increases on average by 5%. Figure 4.6 shows the results of the pred-

icate clustering approach, using predicate matrices with 4 rows, 5 rows and 5 rows with weights, when query states exist in the profile (Figure 4.6a) or do not exist (Figure 4.6b), for the same synthetic data set. The Jaccard coefficient increases at around 10 to 15% for correlated preferences, and on average 5% when a query does not exist in the profile. Clearly, in general, there is a trade-off between the number of produced rankings (i.e. the number of produced clusters) and the quality of the interest scores. In general, the predicate approach results in more accurate top-k rankings, however, the number of scores we maintain for each tuple is larger.

Figure 4.5b shows the results when we use real data sets for both clustering approaches. Using the real data sets, the Jaccard coefficient takes larger values because of the high degree of similarity among user preferences. Again, if a query state does not exist in the profile the results are better, in this case, at around 17%.

Finally, note that when we randomly select a set of preferences to compute the top-20 results, the Jaccard coefficient is nearly equal to zero.

4.6 Summary

In this chapter, we address the problem of finding interesting data items based on contextual preferences that assign interest scores to pieces of data based on context. Given that the database is large and only a few tuples are of interest at any given context, sorting the whole database for each query and context will result in both wasting resources and slow query responses. Thus, we introduced pre-processing steps that can be used to reduce the on-line time for processing each query. In particular, instead of pre-computing scores for all data items under all context states, we have exploited the hierarchical nature of context attributes to identify representative context states. We have also presented a complementary method for grouping contextual preferences according to the similarity of the scores that they produce. This is achieved through a bitmap representation of preferences. Finally, we evaluated our approach using both real and synthetic data sets and presented experimental results showing the quality of the scores attained using our methods.

The results presented in this chapter also appear in [109, 110].

Chapter 5

Personalized Keyword Search through Preferences

Keyword-based search is very popular, because it allows users to express their information needs without either being aware of the underlying structure of the data or using a query language. In relational databases, existing keyword search approaches exploit the database schema (e.g. [63, 12]) or the given database instance (e.g. [17, 57, 72]) to retrieve tuples relevant to the keywords of the query. For example, consider the movie database instance shown in Figure 5.1. Then, the results of the keyword query $Q = \{$*thriller*, *B. Pitt*$\}$ are the *thriller* movies *Twelve Monkeys* and *Seven* both with *B. Pitt*.

Keyword search is intrinsically ambiguous. Given the abundance of available information, exploring the contents of a database is a complex procedure that may return a huge volume of data. Still, users would like to retrieve only a small piece of it, namely the most relevant to their interests. Previous approaches for ranking the results of keyword search include, among others, adapting IR-style relevance ranking strategies [61, 88, 90], authority-based ranking [15] and automated ranking based on workload and data statistics of query answers [28].

In this chapter, we propose personalizing database keyword search, so that, different users receive different results based on their personal interests. To this end, the proposed

model exploits user preferences in ranking keyword results. In this respect, précis queries ([104]) are the most relevant. Précis are keyword queries whose answer is a synthesis of results, containing tuples directly related to the given keywords and tuples implicitly related to them. The database is modeled as a graph in which there is a node for each relation and for each attribute of each relation of the database. Edges have weights that express user preferences. The schema of a query output is determined using the database graph. Each user specifies further degrees of interest and cardinality constraints. While précis provides additional meaning to the results by adding structure, our goal is to use preferences for ranking results.

In our model, preferences express a user *choice* that holds under a specific *context*, where both *context* and *choice* are specified through keywords. For example, consider the following two preferences: ($\{$*thriller*$\}$, *G. Oldman* $\succ$ *W. Allen*) and ($\{$*comedy*$\}$, *W. Allen* $\succ$ *G. Oldman*). The first preference denotes that in the context of *thriller* movies, the user prefers *G. Oldman* over *W. Allen*, whereas the latter, that in the context of *comedies*, the user prefers *W. Allen* over *G. Oldman*. Such preferences may be specified in an ad-hoc manner when the user submits a query or they may be stored in a general user profile. Preferences may also be created automatically based on explicit or implicit user feedback (e.g. [35, 70]) or on the popularity of specific keyword combinations (e.g. [58, 10]). For example, the first preference may be induced by the fact that the keywords *thriller* and *G. Oldman* co-occur in the query log more often than the keywords *thriller* and *W. Allen*.

Given a set of preferences, we would like to personalize a keyword query Q by ranking its results in an order compatible with the order expressed in the user choices for *context* Q. For example, in the results of the query $Q = \{$*thriller*$\}$, movies related to *G. Oldman* should precede those related to *W. Allen*. To formalize this requirement, we consider expansions of query Q with the set of keywords appearing in the user *choice*s for *context* Q. For instance, for the query $Q = \{$*thriller*$\}$, we use the queries $Q_1 = \{$*thriller*, *G. Oldman*$\}$ and $Q_2 = \{$*thriller*, *W. Allen*$\}$. We project the order induced by the user choices among the results of these queries to produce an order among the results of the original query Q.

Since keyword search is often best-effort, given a constraint k on the number of results, we would like to combine the order of results as indicated by the user preferences with their relevance to the query. Besides preferences and relevance, we consider the set of the k results as a whole and seek to increase the overall value of this set to the users. Specifically, we aim at selecting the k most representative among the relevant and preferred results, i.e. these results that both cover different preferences and have different content. In general, result diversification, i.e. selecting items that differ from each other, has been shown to increase user satisfaction [136, 129].

We propose a number of algorithms for computing the top-k results. For generating results that follow the preference order, we rely on applying the *winnow operator* [32, 118] on various levels to retrieve the most preferable choices at each level. Then, we introduce a

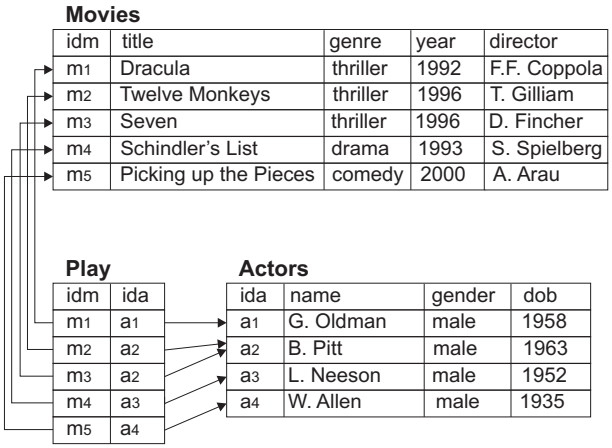


Figure 5.1: Movie database instance.

sharing-results keyword query processing algorithm, that exploits the fact that the results of a keyword query are related with the results of its superset queries, to avoid redundant computations. Finally, we propose an algorithm that works in conjunction with the multi-level winnow and the sharing-results algorithm to compute the top-k representative results.

In a nutshell, in this chapter, we

- propose personalizing keyword search through user preferences and provide a formal model for integrating preferential ranking with database keyword search,
- combine multiple criteria for the quality of the results that include the relevance and the degree of preference of each individual result as well as the coverage and diversity of the set of results as a whole,
- present efficient algorithms for the computation of the top-k representative results.

We have evaluated both the efficiency and effectiveness of our approach. Our performance results show that the sharing-results algorithm improves the execution time over the baseline one by 90%. Furthermore, the overall overhead for preference expansion and diversification is reasonable (around 30% in most cases). Our usability results indicate that users receive results more interesting to them when preferences are used.

The rest of this chapter is organized as follows. In Section 5.1, we introduce our contextual keyword preference model. In Section 5.2, we explore the desired properties of search results and define the top-k representative ones, while in Section 5.3, we propose algorithms for preferential keyword query processing within relational databases. In Section 5.4, we discuss a number of extensions, in Section 5.5, we present our evaluation results and finally, Section 5.6 concludes the chapter.

5.1 Preferential Keyword Model

We start this section with a short introduction to keyword search in databases. Then, we present our model of preferences and personalized keyword search.

5.1.1 Preliminaries

Most approaches to keyword search (e.g. [63, 12]) exploit the dependencies in the database schema for answering keyword queries. Consider a database $\mathcal{R}$ with n relations $R_1, R_2, \ldots, R_n$. The *schema graph* $\mathcal{G}_D$ is a directed graph capturing the foreign key relationships in the schema. $\mathcal{G}_D$ has one node for each relation R_i and an edge $R_i \rightarrow R_j$, if and only if, R_i has a set of foreign key attributes referring to R_j's primary key attributes. We refer to the undirected version of the schema graph as $\mathcal{G}_U$.

Let W be the potentially infinite set of all keywords. A keyword query Q consists of a set of keywords, i.e. $Q \subseteq W$. Typically, the result of a keyword query is defined with regards to *joining trees of tuples* (JTTs), which are trees of tuples connected through primary to foreign key dependencies [63, 12, 17].

Definition 5.1 (Joining Tree of Tuples (JTT))**.** Given an undirected schema graph $\mathcal{G}_U$, a joining tree of tuples (JTT) is a tree T of tuples, such that, for each pair of adjacent tuples t_i, t_j in T, $t_i \in R_i$, $t_j \in R_j$, there is an edge $(R_i, R_j) \in \mathcal{G}_U$ and it holds that $(t_i \bowtie t_j) \in (R_i \bowtie R_j)$.

Total JTT: A JTT T is *total* for a keyword query Q, if and only if, every keyword of Q is contained in at least one tuple of T.

Minimal JTT: A JTT T that is total for a keyword query Q is also *minimal* for Q, if and only if, we cannot remove a tuple from T and get a total JTT for Q.

We can now define the result of a keyword query as follows:

Definition 5.2 (Query Result)**.** Given a keyword query Q, the result $Res(Q)$ of Q is the set of all JTTs that are both total and minimal for Q.

The size of a JTT is equal to the number of its tuples, i.e. the number of nodes in the tree, which is one more than the number of joins. For example, for the database of Figure 5.1, the result of the keyword query $Q = \{$*thriller*, *B. Pitt*$\}$ consists of the JTTs: (i) (m_2, *Twelve Monkeys, thriller, 1996, T. Gilliam*) $-$ (m_2, a_2) $-$ (a_2, *B. Pitt, male, 1963*) and (ii) (m_3, *Seven, thriller, 1996, D. Fincher*) $-$ (m_3, a_2) $-$ (a_2, *B. Pitt, male, 1963*), both of size equal to 3.

5.1.2 Keyword Preference Model

Keyword queries are very general and their result may include a large number of JTTs. We propose personalizing such results by incorporating preferences.

Definition 5.3 (Contextual Keyword Preference)**.** A contextual keyword preference cp is a pair $cp = (C,\ w_i \succ w_j)$, where $C \subseteq W$ and $w_i,\ w_j \in$ W. We also write $w_i \succ_C w_j$.

The intuitive meaning of a contextual keyword preference, or simply preference, is that, when all keywords in context C are present, results involving keyword w_i are preferred over those involving keyword w_j. We refer to $w_i \succ_C w_j$ as the *choice* part of the preference. For example, consider the preference $cp = (\{$*thriller, B. Pitt*$\},$ *T. Gilliam* $\succ$ *D. Fincher*). Preference cp indicates that, in the case of *thrillers* and *B. Pitt*, movies related to *T. Gilliam* are preferred over those related to *D. Fincher*.

Note that we interpret context using AND semantics. This means that a choice holds only if all the keywords of the context part are present (both *thriller* and *B. Pitt* in our example). OR semantics can be achieved by having two or more preferences with the same choice part (for instance, in our example, one for *thriller* and one for *B. Pitt*).

We call the preferences for which the context part is empty, i.e. $C = \{\}$, *context-free* keyword preferences. Context-free keyword preferences may be seen as preferences that hold independently of context. For example, the preference ($\{\}$, *thriller* $\succ$ *drama*) indicates that *thrillers* are preferred over *dramas* unconditionally.

We call the set of all preferences defined by a user, user profile, or simply *profile*. Let P be a profile, we use P_C to denote the set of preferences with context C and W_C to denote the set of keywords that appear in the choices of P_C. We call the keywords in W_C *choice keywords* for C.

We provide next, the formal definition of dominance.

Definition 5.4 (Direct Preferential Domination)**.** Given a keyword query Q and a profile P, let T_i, T_j be two JTTs that are total for Q. We say that T_i directly dominates T_j under P_Q, $T_i \succ_{P_Q} T_j$, if and only if, $\exists w_i$ in T_i, such that, $\nexists w_j$ in T_j with $w_j \succ_Q w_i$ and $w_i, w_j \in W_Q$.

The motivation for this specific formulation of the definition of dominance is twofold. First, we want to favor JTTs that include at least one choice keyword over those that do not include any such keyword. Second, in the case of two JTTs that contain many choice keywords, we want to favor the JTT that contains the most preferred one among them. To clarify this, consider the following example. Assume the query $Q = \{w_q\}$, the choice keywords w_1, w_2, w_3, w_4 and the preferences ($\{w_q\}$, $w_1 \succ w_2$), ($\{w_q\}$, $w_2 \succ w_3$), ($\{w_q\}$, $w_4 \succ w_2$). Let T_1, T_2 be two JTTs in the result set of Q, where T_1 contains, among others, the keywords w_q, w_1, w_3 and T_2 the keywords w_q and w_2. Then, based on Definition 5.4, although T_1 contains the keyword w_3 that is less preferable than w_2 contained in T_2, T_1 directly dominates T_2, because T_1 contains w_1 which is the most preferred keyword among them.

In general, direct dominance $\succ_{P_Q}$ defines a preorder among the JTTs that contain all keywords in Q. Note that it is possible that, for two JTTs T_1, T_2, both $T_1 \succ_{P_Q} T_2$ and $T_2 \succ_{P_Q} T_1$ hold. For instance, in the above example, assume T_1 with w_q and w_1 and T_2 with w_q and w_4. We consider such JTTs to be equally preferred. It is also possible that

neither $T_1 \succ_{P_Q} T_2$ nor $T_2 \succ_{P_Q} T_1$ holds. This is the case when none of the JTTs contain any choice keywords. Such JTTs are incomparable; we discuss next how we can order them.

5.1.3 Extending Dominance

Definition 5.4 can be used to order by dominance those JTTs in the query result that contain choice keywords. For example, given the preference ($\{$*thriller*$\}$, *F. F. Coppola* $\succ$ *T. Gilliam*), for the query $Q = \{$*thriller*$\}$, the JTT $T_1 =$ (m_1, *Dracula, thriller, 1992, F. F. Coppola*) directly dominates the JTT $T_2 =$ (m_2, *Twelve Monkeys, thriller, 1996, T. Gilliam*). However, we cannot order results that may contain choice keywords indirectly through joins. For example, given the preference ($\{$*thriller*$\}$, *G. Oldman* $\succ$ *B. Pitt*) and the same query $Q = \{$*thriller*$\}$, now T_1 and T_2 do not contain any choice keywords and thus are incomparable, whereas again T_1 should be preferred over T_2 since it is a thriller movie related to *G. Oldman*, while T_2 is related to *B. Pitt*.

We capture such indirect dominance through the notion of a JTT projection. Intuitively, a JTT T_i indirectly dominates a JTT T_j, if T_i is the projection of some JTT that directly dominates the JTTs whose projection is T_j.

Projected JTT: Assume a keyword query Q and let T_i, T_j be two JTTs. T_j is a projected JTT of T_i for Q, if and only if, T_j is a subtree of T_i that is total and minimal for Q, that is, $T_j \in Res(Q)$. The set of the projected JTTs of T_i for Q is denoted by $project_Q(T_i)$.

For example, assume the query $Q = \{$*thriller*$\}$. The JTT (m_1, *Dracula, thriller, 1992, F. F. Coppola*) is a projected JTT of (m_1, *Dracula, thriller, 1992, F. F. Coppola*) $-$ (m_1, a_1) $-$ (a_1, *G. Oldman, male, 1958*) for Q.

We can construct the projected JTTs of a JTT T by appropriately removing nodes from T as follows. A leaf node of T is called *secondary* with respect to Q, if it contains a keyword in Q that is also contained in some other node of T. All projected JTTs for T can be produced from T by removing secondary nodes one by one till none remains.

The following set is useful. It contains *exactly* the minimal JTTs that include all keywords in Q and at least one keyword in W_Q.

Definition 5.5 (Preferential Query Result)**.** Given a keyword query Q and a profile P, the preferential query result $PRes(Q, P)$ is the set of all JTTs that are both total and minimal for at least one of the queries $Q \bigcup \{w_i\}$, $w_i \in W_Q$.

Now, we can define indirect dominance as follows:

Definition 5.6 (Indirect Preferential Domination)**.** Given a keyword query Q and a profile P, let T_i, T_j be two JTTs total for Q. We say that T_i indirectly dominates T_j under P_Q, $T_i \succ\succ_{P_Q} T_j$, if there is a JTT $T_i' \in PRes(Q, P)$, such that, $T_i \in project_Q(T_i')$ and there is no joining tree of tuples $T_j' \in PRes(Q, P)$, such that, $T_j \in project_Q(T_j')$ and $T_j' \succ_{P_Q} T_i'$.

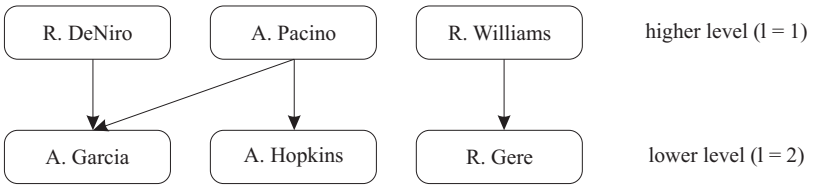


Figure 5.2: The graph of choices $G_{P_{\{thriller,F.F.Coppola\}}}$.

Note that the indirect dominance relation is not a superset of direct dominance, that is, $T_i \succ_{P_Q} T_j \nRightarrow T_i \succ\succ_{P_Q} T_j$. To see this, consider the case where T_i contains a choice keyword that precedes those in T_j but T_j belongs to the project of a JTT that contains an even more preferred keyword.

Our goal in defining indirect preferential dominance is to impose an ordering over the results that will follow the preferences given by the users exactly. Thus, a result that is even only "distantly" related to a choice keyword (i.e. through many joins) is preferred over a result that is more closely related to a less preferred choice keyword. We shall introduce issues of relevance and multi-criteria ranking later in the chapter.

5.1.4 Processing Dominance

Given a query Q, we would like to generate its results in order of indirect dominance. To achieve this, we use the fact that, in general, the trees in the result of $Q \bigcup \{w_i\}$ directly dominate the trees in the result of $Q \bigcup \{w_j\}$, for $w_i \succ_Q w_j$. This suggests that the order for generating the results for a query Q should follow the order $\succ_Q$ among the choice keywords in W_Q. We describe next, how to organize the choice keywords to achieve this.

Let P be a profile, C a context and P_C the related contextual preferences in P. We organize the choice keywords in W_C using a directed graph G_{P_C} for P_C, referred to as *graph of choices* for P_C. G_{P_C} has one node for each keyword $w_i \in W_C$ and an edge from the node representing w_i to the node representing w_j, if and only if, it holds that $w_i \succ_C w_j$ and $\nexists w_r$, such that, $w_i \succ_C w_r$ and $w_r \succ_C w_j$. For example, consider the preferences for $C = \{$*thriller, F. F. Coppola*$\}$: $cp_1 = (C$, *R. DeNiro* $\succ$ *A. Garcia*$)$, $cp_2 = (C$, *A. Pacino* $\succ$ *A. Garcia*$)$, $cp_3 = (C$, *A. Pacino* $\succ$ *A. Hopkins*$)$ and $cp_4 = (C$, *R. Williams* $\succ$ *R. Gere*$)$. The graph of choices for this set of preferences is depicted in Figure 5.2.

To extract from G_{P_C} the set of the most preferred keywords, we apply the *multiple level winnow operator* [32, 118]. This operator retrieves the keywords appearing in G_{P_C} in order of preference. Specifically, at level 1, $win_{P_C}(1) = \{w_i \in W_C \mid \nexists w_j \in W_C, w_j \succ_C w_i\}$. For subsequent applications at level l, $l > 1$, it holds, $win_{P_C}(l) = \{w_i \in W_C \mid \nexists w_j \in (W_C - \bigcup_{r=1}^{l-1} win_{P_C}(r))$ with $w_j \succ_C w_i\}$.

In the following, we assume that the preference relation $\succ_C$ defined over the keywords in W_C is a strict partial order. This means that it is irreflexive, asymmetric and transitive. Irreflexivity and asymmetry are intuitive, while transitivity allows users to define priorities among keywords without the need of specifying relationships between all possible pairs. Strict partial order ensures that there are no cycles in preferences, since that would violate

Multiple Level Winnow Algorithm

Input: A graph of choices $G_{P_C} = (V_G, E_G)$.

Output: The sets $win_{P_C}(l)$ for the levels l.

```
1: Begin
2: winnow_result: empty list;
3: l = 1;
4: while V_G not empty do
5:    for all w_i ∈ V_G with no incoming edges in E_G do
6:       win_{P_C}(l) = win_{P_C}(l) ⋃ {w_i};
7:    Add win_{P_C}(l) to winnow_result;
8:    V_G = V_G − win_{P_C}(l);
9:    for all edges e = (w_i, w_j) with w_i in win_{P_C}(l) do
10:      E_G = E_G − e;
11:   l++;
12: return winnow_result;
13: End
```

Algorithm 13: Multiple Level Winnow Algorithm

irreflexivity.

Since the relation $\succ_C$ is acyclic, this ordering of keywords corresponds to a topological sort of G_{P_C}. Therefore, we traverse the graph of choices G_{P_C} in levels (Algorithm 13) and at each level, we return the keywords of the nodes with no incoming edges. For example, consider the graph of choices of Figure 5.2 for $C = \{$*thriller, F. F. Coppola*$\}$. Then, $win_{P_C}(1) = \{$*R. DeNiro, A. Pacino, R. Williams*$\}$, while $win_{P_C}(2) = \{$*A. Garcia, A. Hopkins, R. Gere*$\}$.

It is useful to define the following: Let T be a JTT that belongs to $PRes(Q, P)$. We associate with T an order $dorder(T, Q, P)$ that encapsulates its preference order with regards to Q and P and is equal to the minimum winnow level l over all choice keywords $w_i \in W_Q$ that appear in T. Then:

Property 5.1. *Let T_i, T_j be two JTTs, T_i, $T_j \in PRes(Q, P)$, such that, $dorder(T_i, Q, P) < dorder(T_j, Q, P)$. Then, T_j does not directly dominate T_i under P_Q.*

Proof. For the purpose of contradiction, assume that $T_j \succ_{P_Q} T_i$. Then, $\exists w_j$ in T_j, such that, $\nexists w_i$ in T_i with $w_i \succ_Q w_j$, which means that $dorder(T_i, Q, P) \geq dorder(T_j, Q, P)$, which is a contradiction. ∎

Thus, by executing the queries $Q \bigcup \{w_1\}, \ldots, Q \bigcup \{w_m\}$, where $\{w_1, \ldots, w_m\}$ are the keywords retrieved by the multiple level winnow operator, in that order, we retrieve the JTTs of $PRes(Q, P)$ in an order compatible with the direct dominance relation among them. Given, for example, the query $Q = \{$*thriller, F. F. Coppola*$\}$ and the preferences cp_1, cp_2, cp_3 and cp_4, we report first the JTTs in the results of $Q \cup \{$*R. DeNiro*$\}$, $Q \cup \{$*A. Pacino*$\}$, $Q \cup \{$*R. Williams*$\}$ and then, those for $Q \cup \{$*A. Garcia*$\}$, $Q \cup \{$*A. Hopkins*$\}$, $Q \cup \{$*R. Gere*$\}$.

By taking the projection of these JTTs in that order, and removing duplicate appearances of the same trees, we take results in $Res(Q)$ in the correct indirect dominance order. Note that a projected result may appear twice as output since it may be related indirectly, i.e. through joins, with more than one choice keyword.

To see that by projecting the JTTs, we get the results in $Res(Q)$ ordered by indirect dominance, let T be a JTT that belongs to $Res(Q)$. We define the indirect order of T, $iorder(T, Q, P)$, to capture its indirect dominance with respect to Q as follows: $iorder(T, Q, P)$ is the minimum $dorder(T', Q, P)$ among all T', such that, $T \in project_Q(T')$ and ∞ if there is no such T'. It holds:

Theorem 5.1. *Let T_i, T_j be two JTTs, T_i, $T_j \in Res(Q)$, such that, $iorder(T_i, Q, P) > iorder(T_j, Q, P)$. Then, T_j does not indirectly dominate T_i under Q.*

Proof. Assume that $T_j \succ\succ_{P_Q} T_i$. Then $\exists T'_j \in PRes(Q, P)$, such that, $T_j \in project_Q(T'_j)$ and $\nexists T'_i \in PRes(Q, P)$, such that, $T_i \in project_Q(T'_i)$ with $T'_i \succ_{P_Q} T'_j$. Since T_i is a subtree of T'_i, $\neg(T_i \succ_{P_Q} T'_j)$ (1). Also, since $iorder(T_i, Q, P) > iorder(T_j, Q, P)$ and $T_j \in project_Q(T'_j)$, T'_j cannot contain any keyword that is preferred over the keywords of T_i. Therefore, $\neg(T'_j \succ_{P_Q} T_i)$ (2). Since T'_j contains at least one choice keyword, (1) and (2) cannot hold simultaneously, which is a contradiction. ∎

Note here that there may be results in $Res(Q)$ that we do not get by projection. Those do not indirectly dominate any result but are indirectly dominated by those that we have gotten by projection.

Theorem 5.2. *Let $S = \bigcup_r project_Q(T_r)$, $\forall T_r \in PRes(Q, P)$, and T_i be a JTT, such that, $T_i \in Res(Q) \backslash S$. Then, $\forall T_j \in S$, it holds that (i) $T_j \succ\succ_{P_Q} T_i$ and (ii) $\neg(T_i \succ\succ_{P_Q} T_j)$.*

Proof. Since $T_i \notin S$, there is no T'_i, $T_i \in project_Q(T'_i)$, such that, T'_i contains a choice keyword of W_Q. However, for every $T_j \in S$ there is at least one T'_j, $T_j \in project_Q(T'_j)$, such that, T'_j contains at least a choice keyword of W_Q. Therefore, according to Definition 5.6, both (i) and (ii) hold. ∎

We can present to the user the projected result or the original JTT in $PRes(Q, P)$, which is not minimal but provides an explanation of why its projected tree in $Res(Q)$ was ordered this way. For instance, for the query $Q = \{thriller\}$, the preference ($\{thriller\}$, *G. Oldman* $\succ$ *B. Pitt*) and the database instance in Figure 5.1, we could either present to the user as top result the JTT (m_1, *Dracula, thriller, 1992, F. F. Coppola*) − (m_1, a_1) − (a_1, *G. Oldman, male, 1958*) that belongs to $PRes(Q, P)$ or its projected JTT (m_1, *Dracula, thriller, 1992, F. F. Coppola*) that belongs to $Res(Q)$.

5.2 Top-k Personalized Results

In general, keyword search is best effort. For achieving useful results, dominance needs to be combined with other criteria. We distinguish between two types of properties that

affect the goodness of the result: (i) properties that refer to each individual JTT in the result and (ii) properties that refer to the result as a whole. The first type includes preferential dominance and relevance, while the latter includes coverage of user interests and diversity.

5.2.1 Result Goodness

Each individual JTT T total for a query Q is characterized by its dominance with regards to a profile, denoted $iorder(T, Q, P)$. In addition, there has been a lot of work on ranking JTTs based on their relevance to the query. A natural characterization of the relevance of a JTT (e.g. [63, 12]) is its size: the smaller the size of the tree, the smaller the number of the corresponding joins, thus the largest its relevance. The relevance of a JTT can also be computed based on the importance of its tuples. For example, [17] assigns scores to JTTs based on the prestige of their tuples, i.e. the number of their neighbors or the strength of their relationships with other tuples, while [61] adapts IR-style document relevance ranking. In the following, we do not restrict to a specific definition of relevance, but instead just assume that each individual JTT T is also characterized by a degree of relevance, denoted $relevance(T, Q)$.

Apart from properties of each individual JTT, to ensure user satisfaction by personalized search, it is also important for the whole set of results to exhibit some desired properties. In this work, we consider covering many user interests and avoiding redundant information.

To understand coverage, consider the graph of choices in Figure 5.2. JTTs for the query Q ={*thriller, F. F. Copolla*} that include *R. DeNiro* and *R. Williams* have the same degree of dominance and assume, for the purposes of this example, that they also have the same relevance. Still, we would expect that a good result does not only include JTTs (i.e. movies) that cover the preference on *R. DeNiro* but also JTTs that cover the preference on *R. Williams* and perhaps other choices as well. To capture this requirement, we define the *coverage* of a set S of JTTs with regards to a query Q as the percentage of choice keywords in W_Q that appear in S. Formally:

Definition 5.7 (Coverage)**.** Given a query Q, a profile P and a set $S = \{T_1, \ldots, T_z\}$ of JTTs that are total for Q, the coverage of S for Q and P is defined as:

$$coverage(S, Q, P) = \frac{|\bigcup_{i=1}^{z}(W_Q \cap keywords(T_i))|}{|W_Q|},$$

where $keywords(T_i)$ is the set of keywords in T_i.

High coverage ensures that the user will find many interesting results among the retrieved ones. However, many times, two JTTs may contain the same or very similar information, even if they are computed for different choice keywords. To avoid such redundant information, we opt to provide users with results that exhibit some diversity, i.e. they do not contain overlapping information. For quantifying the overlap between

two JTTs, we use a Jaccard-based definition of distance, which measures dissimilarity between the tuples that form these trees. Given two JTTs T_i, T_j consisting of the sets of tuples A, B respectively, the distance between T_i and T_j is:

$$d(T_i, T_j) = 1 - \frac{|A \cap B|}{|A \cup B|}.$$

We have considered other types of distances as well, but this is simple, relatively fast to compute and provides a good indication of the overlapping content of the two trees.

To measure the overall diversity of a set of JTTs, we next define their set diversity based on their distances from each other. A number of different definitions for set diversity have been proposed in the context of recommender systems; here we model diversity as the average distance of all pairs of elements in the set [133].

Definition 5.8 (Set Diversity)**.** Given a set S of z JTTs, $S = \{T_1, \ldots, T_z\}$, the set diversity of S is:

$$diversity(S) = \frac{\sum_{i=1}^{z} \sum_{j>i}^{z} d(T_i, T_j)}{(z-1)z/2}.$$

To summarize, a "good" result S for a query Q includes JTTs that are preferred and relevant, covers many choices and is diverse.

5.2.2 Top-k Result Selection

Given a restriction k on the size of the result, we would like to provide users with k highly preferable and relevant results that also as a whole cover many of their choices and exhibit low redundancy. To achieve this, we resort to the following algorithm that offers us the flexibility of fine-tuning the importance of each of the criteria in selecting the top-k results.

For a query Q, we use $Res_s(Q)$ to denote the set of JTTs with relevance greater than a threshold s. Given a query Q and a profile P, let l be the maximum winnow level. For $1 \leq r \leq l$, let $Z^r = \bigcup_{w_j \in win_{P_Q}(r)} Res_s(Q \cup \{w_j\})$. Also, let $Z^{l+1} = Res_s(Q) \setminus \bigcup_{T_e \in PRes(Q,P)} project_Q(T_e)$. We want more preferred keywords, that is, the ones corresponding to small winnow values, to contribute more trees to the top-k results than less preferred ones. The number of trees offered by each level i is captured by $\mathcal{F}(i)$, where $\mathcal{F}$ is a monotonically decreasing function with $\sum_{i=1}^{l+1} \mathcal{F}(i) = k$. Each Z^i contributes $\mathcal{F}(i)$ JTTs. For $1 \leq i \leq l$, the contributed JTTs are uniformly distributed among the keywords of level i to increase coverage.

Among the many possible combinations of k trees that satisfy the constraints imposed by $\mathcal{F}$, we choose the one with the most diverse results. Next, we define the top-k JTTs.

Definition 5.9 (Top-k JTTs)**.** Given a keyword query Q, a profile P, a relevance threshold s and the sets of results $\{Z^1, \ldots, Z^l, Z^{l+1}\}$ with $|Z^1| + \ldots + |Z^l| + |Z^{l+1}| = m$, the top-$k$ JTTs, $k < m$, is the set S^* for which:

$$S^* = \underset{\substack{S \subseteq \bigcup_{i=1}^{l+1} Z^i \\ |S|=k}}{\text{argmax}} \; diversity(S),$$

such that, Z^i contributes $\mathcal{F}(i)$ JTTs to S^*, which, for $1 \leq i \leq l$, are uniformly distributed among the keywords of winnow level i and $\mathcal{F}$ is a monotonically decreasing function with $\sum_{i=1}^{l+1} \mathcal{F}(i) = k$.

There are two basic tuning parameters: function $\mathcal{F}$ and threshold s. Dominance, coverage and relevance depend on how quickly $\mathcal{F}$ decreases. A high decrease rate leads to keywords from fewer winnow levels contributing to the final result. This means that coverage will generally decrease. However, at the same time, the average dominance will increase, since the returned results correspond to high winnow levels only. For example, if a user is primarily interested in dominant results, we retrieve k JTTs corresponding to keywords retrieved by $win_{P_Q}(1)$ by setting, for example, $\mathcal{F}(1) = k$, and $\mathcal{F}(i) = 0$, for $i > 1$. A low decrease rate of $\mathcal{F}$ means that less trees will be retrieved from each winnow level, so we can retrieve the most relevant ones. Relevance is also calibrated through the selection of the relevance threshold, s. If relevance is more important than dominance, a large value for the relevance threshold in conjunction with an appropriate $\mathcal{F}$ will result in retrieving the k JTTs that have the largest degrees of relevance, including those in Z^{l+1} that do no have any relation with any choice keyword. Diversity is calibrated through s that determines the number m of candidate trees out of which to select the k most diverse ones.

5.3 Query Processing

In this section, we present our algorithms for processing personalized keyword queries. Section 5.3.1 presents some background, while in Section 5.3.2, we first present a baseline algorithm for processing keyword queries and then introduce an enhancement that reuses computational steps to improve performance. In Section 5.3.3, we propose an algorithm for computing top-k results.

5.3.1 Background

We use our movies example (Figure 5.1) to briefly describe basic ideas of existing keyword query processing. For instance, consider the query $Q = \{$*thriller*, *B. Pitt*$\}$. The corresponding result consists of the JTTs: (i) (m_2, *Twelve Monkeys, thriller, 1996, T. Gilliam*) $-$ (m_2, a_2) $-$ (a_2, *B. Pitt, male, 1963*) and (ii) (m_3, *Seven, thriller, 1996, D. Fincher*) $-$ (m_3, a_2) $-$ (a_2, *B. Pitt, male, 1963*). Each JTT corresponds to a tree at schema level. For example, both of the above trees correspond to the schema level tree $Movies^{\{thriller\}} - Play^{\{\}} - Actors^{\{B.Pitt\}}$, where each R_i^X consists of the tuples of R_i that contain all keywords of X and no other keyword of Q. Such sets are called *tuple sets* and the schema level trees *joining trees of tuple sets* (JTSs).

Several algorithms in the research literature aim at constructing such trees of tuple sets for a query Q as an intermediate step of the computation of the final results (e.g. [63, 12]). In the following, we adopt the approach of [63], in which all JTSs with size up

to s are constructed (in this case, a JTT's size determines its relevance). In particular, given a query Q, all possible tuple sets R_i^X are computed, where $R_i^X = \{t \mid t \in R_i \wedge \forall w_x \in X, t \text{ contains } w_x \wedge \forall w_y \in Q \backslash X, t \text{ does not contain } w_y\}$. After selecting a random query keyword w_z, all tuple sets R_i^X for which $w_z \in X$ are located. These are the initial JTSs with only one node. Then, these trees are expanded either by adding a tuple set that contains at least another query keyword or a tuple set for which $X = \{\}$ (free tuple set). These trees can be further expanded. JTSs that contain all query keywords are returned, while JTSs of the form $R_i^X - R_j^{\{\}} - R_i^Y$, where an edge $R_j \rightarrow R_i$ exists in the schema graph, are pruned, since JTTs produced by them have more than one occurrence of the same tuple for every instance of the database.

5.3.2 Processing Preferential Queries

In this section, we present algorithms for computing the preferential results of a query, ranked in an order compatible with preferential dominance.

Baseline Approach

The *Baseline JTS Algorithm* (Algorithm 14) constructs in levels the set of JTSs for the queries $Q \cup \{w_i\}$, $\forall w_i \in win_{P_Q}(l)$, starting with $l = 1$, i.e. the level with the most preferred keywords. This way, all JTTs constructed for JTSs produced at level l are retrieved before the JTTs of the trees of tuple sets produced at level l+1. Algorithm 14 terminates when all the JTSs for queries $Q \cup \{w_i\}$, $\forall w_i \in W_{P_Q}$, have been computed. (In Algorithm 14, we use the notation $keys(B)$ to refer to the query keywords contained in a JTS B.)

Based on the completeness theorem of the algorithm introduced in [63] for computing the JTSs, Theorem 5.3 proves the completeness of Algorithm 14.

Theorem 5.3 (Completeness)**.** *Every JTT of size s_i that belongs to the preferential query result of a keyword query Q is produced by a JTS of size s_i that is constructed by the Baseline JTS Algorithm.*

Proof. Given a query Q and a profile P, the *Baseline JTS Algorithm* constructs independently the JTSs for each query $Q \cup \{w_t\}$, $\forall w_t \in W_{P_C}$ (lines 8-27). Since for each query the algorithm returns the trees of tuple sets that are capable to construct every JTT that belongs to the corresponding result, every JTT that belongs to $PRes(Q, P)$ is produced by the JTSs constructed by Algorithm 14 as well. ∎

Result Sharing

Based on the observation that the JTSs for Q may already contain in their tuple sets the additional keyword w_t of a query $Q_t \in KQ$, where KQ contains the queries $Q_t = Q \cup \{w_t\}$, $\forall w_t \in W_{P_Q}$, we employ such trees to construct those for Q_t. To do this, the

Baseline JTS Algorithm

Input: A query Q, a profile P, a schema graph $\mathcal{G}_U$ and a size s.

Output: A list JTList of JTSs with size up to s for the queries $Q \bigcup \{w_i\}$, $\forall w_i \in W_{P_Q}$.

```
1: Begin
2: Queue: queue of JTSs;
3: JTList: empty list;
4: l = 1;
5: while unmarked keywords exist in W_{P_Q} do
6:    Compute the set of keywords win_{P_Q}(l);
7:    for each w_z ∈ win_{P_Q}(l) do
8:       Mark w_z;
9:       Compute the tuple sets R_i^X for Q ⋃ {w_z};
10:      Select a keyword w_t ∈ Q ⋃ {w_z};
11:      for each R_i^X, 1 ≤ i ≤ n, such that, w_t ∈ X do
12:         Insert R_i^X into Queue;
13:      while Queue ≠ ∅ do
14:         Remove the head B from Queue;
15:         if B satisfies the pruning rule then
16:            Ignore B;
17:         else if keys(B) = Q ⋃ {w_z} then
18:            Insert B into JTList;
19:         else
20:            for each R_i^X, such that, there is an R_j^Y in B and R_i is adjacent to R_j in G_U do
21:               if (X = {} OR X − keys(B) ≠ ∅) AND (size of B < s) then
22:                  Expand B to include R_i^X;
23:                  Insert the updated B into Queue;
24:   l++;
25: return JTList;
26: End
```

Algorithm 14: Baseline JTS Algorithm

Sharing JTS Algorithm (Algorithm 15) constructs first the JTSs for Q using a selected keyword $w_r \in Q$ based on the tuple sets R_i^X for Q (lines 3-5). Then, for each Q_t, we recompute its tuple sets by partitioning each R_i^X for Q into two tuple sets for Q_t: R_i^X that contains the tuples with only the keywords X and $R_i^{X \cup \{w_t\}}$ that contains the tuples with only the keywords $X \cup \{w_t\}$ (lines 11-13). Using the JTSs for Q and the tuple sets for Q_t, we produce all combinations of trees of tuple sets (lines 14-17) that will be used next to construct the final JTSs for Q_t. For example, given the JTS for Q R_i^X - R_j^Y, we produce the following JTSs for Q_t: R_i^X - R_j^Y, $R_i^{X \cup \{w_t\}}$ - R_j^Y, R_i^X - $R_j^{Y \cup \{w_t\}}$ and $R_i^{X \cup \{w_t\}}$ - $R_j^{X \cup \{w_t\}}$. Note that, such a JTS is constructed only if all of its tuples sets are non-empty. The JTSs that contain all keywords of Q_t are returned. The rest of them are expanded as in Algorithm 14 (lines 33-42).

Since for a query Q, Algorithm 14 does not construct JTSs of the form $R_i^{\{w_k\}}$ - $R_j^{\{w_k\}}$, the procedure described above does not construct for Q_t JTSs of the form $R_i^{\{w_k\}}$ - $R_j^{\{w_k,w_t\}}$. The same also holds for the JTSs that connect $R_i^{\{w_k\}}$, $R_j^{\{w_k,w_t\}}$ via free tuple sets. To overcome this, we construct all such trees from scratch (lines 18-32) and then expand them as before (lines 33-42). Theorem 5.4 proves the completeness of Algorithm 15.

Theorem 5.4 (Completeness)**.** *Every JTT of size s_i that belongs to the preferential query result of a keyword query Q is produced by a JTS of size s_i that is constructed by the Sharing JTS Algorithm.*

Proof. Let Q be a query, P a profile and S the set of JTSs, such that, each JTT in $PRes(Q, P)$ can be produced by a JTS in S. S is divided into two sets S_1 and S_2, such that, $S_1 \cap S_2 = \emptyset$ and $S_1 \cup S_2 = S$. S_1 consists of all JTSs containing both the tuple sets $R_i^{\{w_r\}}$, $R_j^{\{w_r,w_t\}}$ for a selected keyword $w_r \in Q$, $\forall w_t \in W_{P_Q}$, and S_2 all the rest. With respect to Algorithm 15, JTSs of S_2 are constructed through the lines 3-5, 11-17 and 33-42, while JTSs of S_1 are constructed through the lines 18-42. Therefore, in any case, every JTT in $PRes(Q, P)$, can be produced by a JTS constructed by the *Sharing JTS Algorithm*. ∎

5.3.3 Top-k Query Processing

In the previous section, we introduced the *Sharing JTS Algorithm* that efficiently constructs all JTSs for a query Q. Next, we focus on how to retrieve the top-k results for Q (see Definition 5.9). In general, we use the function $\mathcal{F}$ to determine the number of JTTs each level contributes to the result, thus calibrating preferential dominance, while the specific trees of the result are selected based on their relevance, coverage and diversity.

Relevance is tuned through the maximum size s of the JTSs constructed with regards to Algorithms 14 and 15, while coverage is ensured by selecting trees from each level i, so that, as many keywords as possible are represented in the final result. Concerning diversity, we have to identify the trees with the maximum pair-wise distances.

Given the set $\mathcal{Z} = \bigcup_i Z^i$ of m relevant JTTs, our goal is to produce a new set S, $S \subset \mathcal{Z}$, with the k most diverse JTTs, $k < m$, such that, Z^i contributes $\mathcal{F}(i)$ trees. The problem of selecting the k items having the maximum average pair-wise distance out of m items is similar to the *p-dispersion-sum* problem. This problem as well as other variations of the general p-dispersion problem (i.e. select p out of m points, so that, the minimum distance between any two pairs is maximized) have been studied in operations research and are in general known to be NP-hard [43].

A brute-force method to locate the k most diverse JTTs of $\mathcal{Z} = \bigcup_i Z^i$, $|\mathcal{Z}| = m$, is to first produce all $\binom{m}{k}$ possible combinations of trees and then pick the one with the maximum set diversity out of those that satisfy the constraints of Definition 5.9. The complexity of this process is exponential and therefore, the computational cost is too high even for low values of m and k. A number of lower-complexity heuristics have been

Sharing JTS Algorithm

Input: A profile P, a set of queries KQ of the form $Q_t = Q \cup \{w_t\}$, $\forall w_t \in W_{P_Q}$, a schema graph $\mathcal{G}_U$ and a size s.

Output: A list JTList of JTSs with size up to s for the queries in KQ.

1: **Begin**
2: $Queue_1$, Q': queues of JTSs;
3: $JTS^Q, JTSList$: empty lists;
4: Compute the tuple sets R_i^X with regards to Q;
5: Select a keyword $w_r \in Q$;
6: Construct and insert to JTS^Q the JTSs of Q; /* as steps 9-27 of the Baseline JTS Algorithm */
7: $l = 1$;
8: **while** unmarked keywords exist in W_{P_Q} **do**
9: Compute the set of keywords $win_{P_Q}(l)$;
10: **for** each $Q_t \in KQ$, such that, $w_t \in win_{P_Q}(l)$ **do**
11: Mark w_t;
12: **for** each R_i^X, $1 \leq i \leq n$, computed for Q **do**
13: Construct the tuple sets R_i^X and $R_i^{X \cup \{w_t\}}$ for Q_t;
14: **for** each JTS in JTS^Q **do**
15: Construct all combinations of trees of tuple sets by replacing the tuple sets of Q with the relative tuple sets of Q_t;
16: Insert those JTSs into $Queue_1$;
17: **for** each $R_i^{\{w_r\}}$, $1 \leq i \leq n$, computed for Q_t **do**
18: Insert $R_i^{\{w_r\}}$ into $Queue_2$;
19: **while** $Queue_2 \neq \emptyset$ **do**
20: Remove the head B from $Queue_2$;
21: **for** each R_i^X, such that, there is an R_j^Y in B and R_i is adjacent to R_j in $\mathcal{G}_U$ **do**
22: **if** $X = \{w_r, w_t\}$ AND size of $B < s - 1$ **then**
23: Expand B to include R_i^X;
24: Insert the updated B into $Queue_1$;
25: **else if** $X = \{\}$ AND size of $B < s - 1$ **then**
26: Expand B to include R_i^X;
27: Insert the updated B into $Queue_2$;
28: **while** $Queue_1 \neq \emptyset$ **do**
29: Remove the head B from $Queue_1$;
30: **if** T satisfies the pruning rule **then**
31: Ignore B;
32: **else if** $keys(B) = Q \bigcup \{w_t\}$ **then**
33: Insert B into $JTList$;
34: **else**
35: /* as steps 20-25 of the Baseline JTS Algorithm */
36: l++;
37: **return** $JTlist$;
38: **End**

Algorithm 15: Sharing JTS Algorithm

proposed to locate subsets of elements (e.g. in [43]). In this work, we use the following variation: we construct a diverse subset of JTTs based on the tree-set distance.

Definition 5.10 (Tree-Set Distance)**.** Given a JTT T and a set of JTTs $S = \{T_1, \ldots, T_z\}$, the tree-set distance between T and S is:

$$dist(T, S) = \min_{1 \leq i \leq z} d(T, T_i).$$

Initially, we consider an empty set S. We first add to S the two furthest apart elements of Z^1. Then, we incrementally construct S by selecting trees of $Z^1 \backslash S$ based on their tree-set distance from the trees already in S. In particular, we compute the distances $dist(T_i, S)$, $\forall T_i \in Z^1 \backslash S$ and add to S the tree with the maximum corresponding distance. When $\frac{\mathcal{F}(1)}{|win_{P_Q}(1)|}$ trees have been added to S for a keyword in $win_{P_Q}(1)$, we exclude JTTs computed for that keyword from Z^1. Then, we proceed by selecting trees from $Z^2 \backslash S$ until another $\mathcal{F}(2)$ trees have been added to S and so on.

We can further reduce the number of performed operations based on the observation that after the insertion of a tree T to S, the distances of all other trees that have not yet entered the diverse results from S', $S' = S \cup \{T\}$, are affected only by the presence of T. This leads us to the following proposition:

Property 5.2. *Given a JTT T_i and two sets of JTTs S and S', $S' = S \cup \{T_i\}$, it holds that:*

$$dist(T_j, S') = \min\{dist(T_j, S), d(T_j, T_i)\}.$$

The above process is shown in Algorithm 16. Observe that, threshold-based top-k algorithms (e.g. [49]) cannot be applied to construct diverse subsets of JTTs, since the $k-1$ most diverse trees of $\mathcal{Z}$ are not necessarily a subset of its k most diverse ones.

5.4 Extensions

In this section, we consider extending the preference model by relaxing its context part and allowing more keywords in its choice part. We also discuss a simple approach for deriving preferences.

5.4.1 Relaxing Context

For a profile P and a query Q, the associated set of preferences P_Q may be empty, that is, there may be no preferences for Q. In this case, we can use for personalization those preferences whose context is more general than Q, i.e. their context is a subset of Q.

Definition 5.11 (Relaxed Context)**.** Given a query Q and a profile P, a set $C \subset Q$ is a relaxed context for Q in P, if and only if, (i) $\exists$ (C, *choice*) $\in P$ and (ii) $\nexists$ (C', *choice'*) $\in P$, such that, $C' \subset Q$ and $C \subset C'$.

Top-k JTTs Algorithm

Input: The sets of keywords $win_{P_Q}(1), \ldots, win_{P_Q}(l)$ and the sets of JTTs $Z^1, \ldots, Z^l, Z^{l+1}$.
Output: The set S of the top-k JTTs.

```
1: Begin
2: S = ∅;
3: for i = 1; i <= l; i++ do
4:    for each j ∈ win_{P_Q}(i) do
5:       counter(i, j) = F(i) / |win_{P_Q}(i)|;
6: Find the trees T_1, T_2 ∈ Z^1 with the maximum distance;
7: S = S ∪ T_1;
8: S = S ∪ T_2;
9: for i = 1; i <= l + 1; i++ do
10:   for j = 0; j < F(i); j++ do
11:      Find the tree T ∈ Z^i \ S with the maximum dist(T, S);
12:      S = S ∪ T;
13:      if i < l + 1 then
14:         Find the keyword w that T was computed for;
15:         counter(i, w) = counter(i, w) − 1;
16:         if counter(i, w) == 0 then
17:            Remove from Z^i all JTTs computed for w;
18: End
```

Algorithm 16: Top-k JTTs Algorithm

Given a profile P, the *relaxed preferential result* of a query Q is the set of all JTTs that are both total and minimal for at least one of the queries $Q \bigcup \{w_i\}$, $w_i \in W_{P_C}$, where C is a relaxed context for Q. That is, we do not relax the original query Q, but instead, we just use the choice keywords of a relaxed context for Q.

To depict the subset relation among contexts, a lattice representation can be used. Any context-free preference is placed on the top of the lattice. An example is shown in Figure 5.3. For instance, given the preferences of Figure 5.3 and the query $Q = \{$*thriller, F. F. Coppola, R. DeNiro*$\}$, since there is no preference with context equal to Q, the choice keywords of preference cp_9, whose context is a relaxed context for Q, will be used. Note that the context of cp_4 is also more general than Q, but it is not a relaxed context for Q because the context of cp_9 is more specific.

If there is no preference more specific to Q, we finally select the context-free preferences, if any. Finally, note that there may be more than one relaxed context for Q. For instance, for the query $Q = \{$*thriller, S. Spielberg, L. Neeson*$\}$, both $\{$*thriller, S. Spielberg*$\}$ and $\{$*S. Spielberg, L. Neeson*$\}$ are relaxed contexts. In this case, we can use either of them. We could also use more than one relaxed context but this raises semantic issues with regards to how to compose the associated orders between their choice keywords, if they are conflicting, which is an issue beyond the scope of this work.

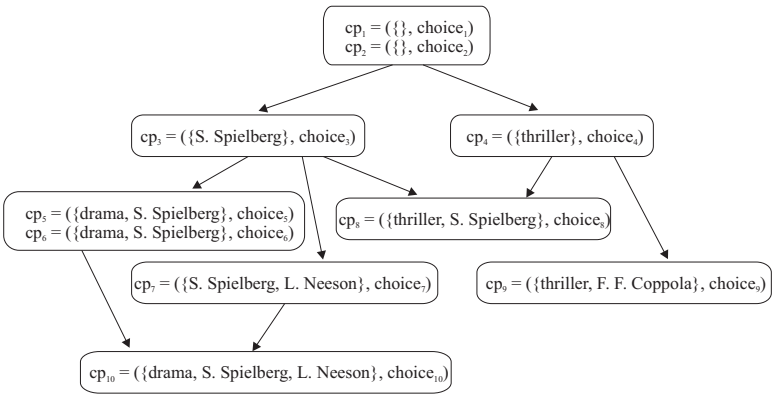


Figure 5.3: Context lattice of preferences.

5.4.2 Multi-Keyword Choices

Our model of preferences supports choices between two keywords. One may think of more complex preferences of the form $(C, choice)$, where $C \subseteq W$ and $choice = (w_{l_1} \wedge \ldots \wedge w_{l_x}) \succ (w_{r_1} \wedge \ldots \wedge w_{r_y})$, w_{l_j}, w_{r_z}, $1 \leq j \leq x$, $1 \leq z \leq y$, $\in W$. We shall refer to such preferences as *composite contextual keyword preferences*. As an example, consider the preference $ccp = (\{comedy,\ W.\ Allen\}, (E.\ Norton \wedge D.\ Barrymore) \succ (B.\ Crystal \wedge D.\ Moore))$. The meaning of ccp is that in the case of *comedy* movies and *W. Allen*, those movies that are related to both *E. Norton* and *D. Barrymore* are preferred over those that are related to both *B. Crystal* and *D. Moore*. Choices can be constructed arbitrarily, in the sense that each choice can have any number of keywords and different number of keywords can be used for the left and the right part, i.e. it may hold that $x \neq y$.

Supporting composite preferences of this form is straightforward. In this case, the preferential result of a query Q is the set of all JTTs that are both total and minimal for at least one of the queries $Q \bigcup W_i$, where W_i is now a set of keywords that appear in one of the parts of a choice for Q. The algorithms of Sections 5.3.2, 5.3.3 can be applied without any modifications. However, to speed up query processing when preferences with composite choices are used, we can further exploit the main idea of the *Sharing JTS Algorithm*. In particular, consider a query Q and two sets of keywords W_1, W_2 that appear in the choices of the relevant to Q preferences. During the searching process, the JTSs of $Q \bigcup W_1$ and $Q \bigcup W_2$ will be computed. Assuming that $W_1 \bigcap W_2 \neq \{\}$, we could first compute the JTSs of $Q \bigcup (W_1 \bigcap W_2)$ and then use them to find the JTSs of $Q \bigcup W_1$ and $Q \bigcup W_2$, instead of computing them from scratch.

5.4.3 Profile Generation

User preferences can either be explicitly provided by the user or be automatically constructed based on the previous user interactions or other available information. Although the focus here is on how to exploit already constructed profiles to personalize keyword database search, we also discuss here a method for potentially inferring contextual keyword preferences in the absence of user input.

Assume that we maintain a log H of the keyword queries submitted to the database. To allow multiple occurrences of the same query Q in H, let us assume that each submitted query is preceded in the log by a unique identifier, that is, H is a set of entries of the form (id, Q) for each submitted query, where id is a unique identifier and Q the content of the query, that is, its keywords. For instance, H ={(id_1, {*thriller, G. Oldman*}), (id_2, {*drama, S. Spielberg*}), (id_3, {*drama, Q. Tarantino*}), (id_4, {*drama, 1993, S. Spielberg*})} (id_5, {*comedy, W. Allen*}), (id_6, {*drama, S. Spielberg*})} is a log of six queries, where, for example, the query {*drama, S. Spielberg*} was submitted twice.

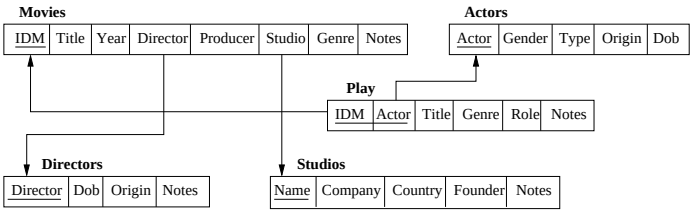
Let W' be a set of keywords, $W' \subseteq W$. We use $freq(W')$ to denote the number of queries in H in which W' appears: $freq(W') = |\{(id, Q) \in H$, such that, $W' \subseteq Q\}|$. For instance, in the example H, $freq(\{drama\}) = 4$. Our underlying assumption is that high popularity of a set W' of keywords, i.e. a large $freq(W')$ value, implies a preference on W', which is an assumption commonly made by many preference learning algorithms [58, 10]. More precisely, when a keyword w_i appears together with a set of other keywords W', i.e. in the same query with them, more frequently than a keyword w_j does, this is considered as an indication that w_i is preferred over w_j in the context of W'. Thus, we create a contextual preference (W', $w_i \succ w_j$), for w_i, $w_j \notin W'$, if $freq(W' \bigcup \{w_i\}) - freq(W' \bigcup \{w_j\}) \geq minf \times |H|$, where $minf < 1$ is a positive constant that tunes the strength of the preferences. For instance, for the example H and $minf = 0.30$, we infer the contextual keyword preference ({*drama*}, *S. Spielberg* $\succ$ *Q. Tarantino*).

Note that the above rule, for context-free preferences, i.e. for $W' = \{\}$, gives us $w_i \succ w_j$ if $freq(\{w_i\}) - freq(\{w_j\}) \geq minf \times |H|$, which simply gives priority to popular keywords over less popular ones. Recall that through context relaxation, context-free preferences will be applied when nothing more specific exists. This means that, for instance, for a query whose keywords have not appear in any of the queries in H, we can use such context-free preferences to personalize it.

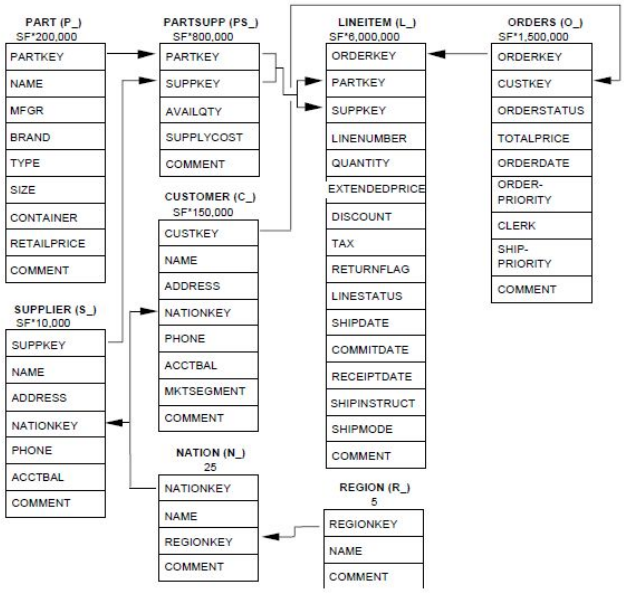
5.5 Evaluation

To evaluate the efficiency and effectiveness of our approach, we conducted a number of experiments, using both real and synthetic datasets: (i) the Movie database [7] (Figure 5.4a) and (ii) the TPC-H database [8] (Figure 5.4b). The Movie dataset consists of nearly 11500 movies, 6800 actors, 3300 directors and 175 studios, while the relation *Play* contains more than 45000 tuples. For the TPC-H database, to experiment with the distribution of each keyword's appearance, we do not use its actual dataset but rather only its schema and we generate data using the following procedure (that was also used in [63]). Each keyword appears in a relation R_i of the database with a probability equal to $\frac{\log(\text{size}(R_i))}{x \cdot \log(y)}$, where size($R_i$) is the cardinality of R_i and y is the cardinality of the largest relation in the database. The lower the value of x, the higher the probability for a keyword to appear in a relationship.

We run our performance experiments for (i) queries of a different size $|q|$, (ii) profiles



(a) Movie schema.



(b) TPC-H schema (*image from www.tpc.org*).

Figure 5.4: Movie and TPC-H database schemas.

with a different number of preferences and thus, a different number of relevant choice keywords $|w|$, (iii) various maximum sizes s for the computed JTTs and (iv) different keyword selectivities. We use MySQL 5.0 to store our data. Our system is implemented in JDK 1.5 and connects to the DBMS through JDBC. We use an Intel Pentium D 3.0GHz PC with 1GB of RAM. The profiles and queries used in our experiments along with the source code and datasets are available for download [4].

5.5.1 Performance Evaluation

In our performance evaluation study, we focus on (i) highlighting the efficiency of the *Sharing JTS Algorithm*, (ii) demonstrating the effectiveness of the *Top-k JTTs Algorithm* and (iii) assessing the overhead of query personalization as well as the reduction in the result size achieved.

Sharing vs. Baseline JTS Algorithm

To illustrate the efficiency of the *Sharing JTS Algorithm* versus the *Baseline* alternative, we measure the execution time and the total number of join operations performed during

Table 5.1: TPC-H dataset: Varying keyword selectivity.

x	s	Time (msec)		Number of joins	
		Baseline	**Sharing**	**Baseline**	**Sharing**
10	3	10.8	2.38	73.7	14.59
	4	68.8	5.31	451.45	87.65
	5	618.99	32.5	3402.5	666.12
8	3	12.19	2.5	87.44	17.29
	4	84.28	5.8	524.1	108.7
	5	701.20	38.19	3877.93	752.75
6	3	14.81	2.74	107.68	22.08
	4	91.64	6.67	605.91	126.22
	5	805.91	50.34	4277.75	950.37

the phase of joining trees of tuple sets expansion.

Figures 5.5a, 5.5b report the execution time and total number of join operations, for the TPC-H database, for $|w| = 10$ when $|q|$ and s vary, while Figures 5.5c, 5.5d show the values of the corresponding measures for $|q| = 3$ and varying $|w|$, s. In all cases, we consider that $x = 10$, which means that the probability of a keyword appearing in the largest relation ($LINEITEM$) is 10%, while for the smallest relation ($REGION$), this probability is around 1%. The *Sharing JTS algorithm* is more efficient, performing only a small fraction of the join operations performed by the *Baseline* one, thus, also requiring much less time. As s increases, the reduction becomes more evident, since the larger this size is, the more the computational steps that are shared. For example, in Figure 5.5a, when $s = 5$, the *Sharing JTS Algorithm* requires only 2.5%-10.5% of the time required by the *Baseline JTS Algorithm*. Observe that, while the number of joins for the *Baseline JTS Algorithm* increases along with $|q|$, it decreases for the *Sharing JTS Algorithm* (Figure 5.5b). This happens because for a larger value of $|q|$, the trees of the preferential result share larger common sub-trees, therefore the *Sharing JTS Algorithm* performs fewer expansions.

To study the impact of keyword selectivity, we also run a set of experiments for $x = 10, 8, 6$ and $s = 3, 4, 5$ for constant values of $|q|$ and $|w|$ ($|q| = 3$ and $|w| = 10$). The results are shown in Table 5.1. For lower values of x, i.e. higher keyword selectivity, both the execution time and join operations increase for both algorithms, since more results exist. In all cases though, the *Sharing JTS Algorithm* outperforms the *Baseline* one. For example, for $x = 8$, the reduction in execution time is around 90%, while the join operations are reduced by 80%.

Similar observations can be made for the Movie database. In this case, we manually picked keywords with various selectivities, trying to construct queries and profiles that lead to results of different sizes and relevance. The *Sharing JTS Algorithm* requires

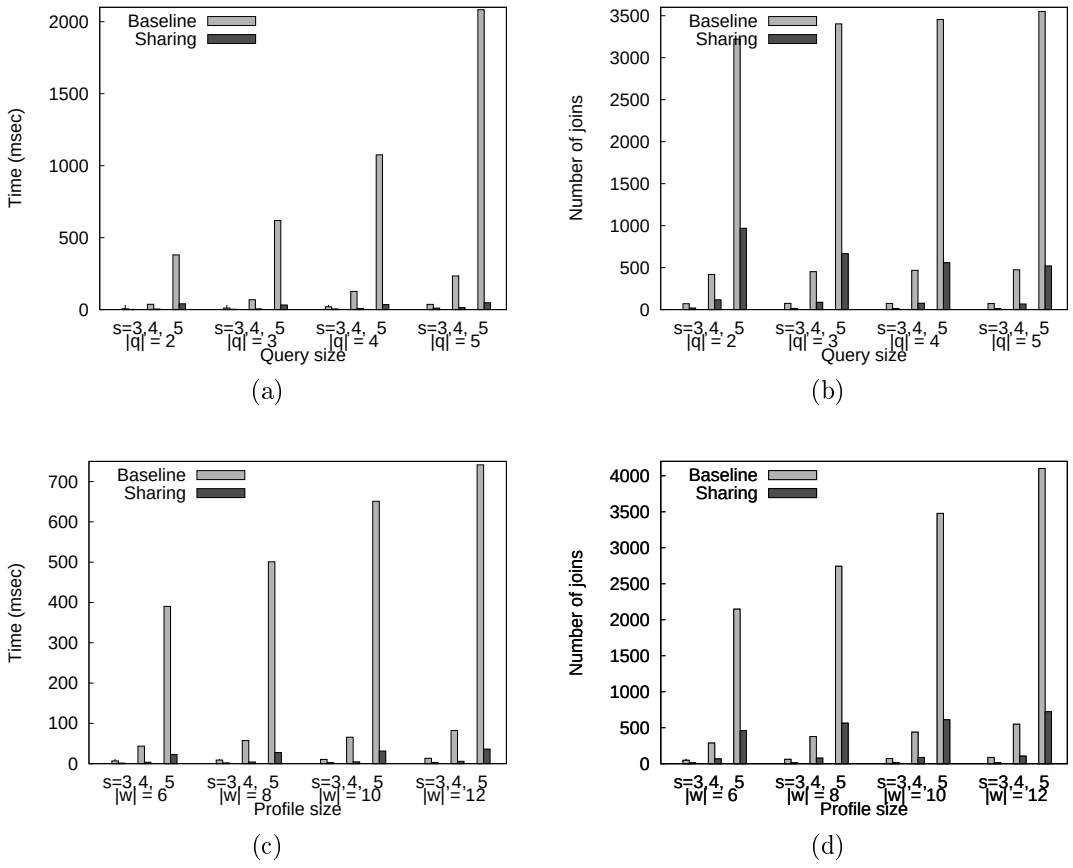


Figure 5.5: TPC-H dataset: Total time for (a) a fixed profile and (c) a fixed query and total number of join operations for (b) a fixed profile and (d) a fixed query.

around 10% of the time required by the *Baseline JTS Algorithm*, while the reduction of join operations during the expansion phase depends on $|q|$ and varies from 90% to 50%. The corresponding results are shown in Figure 5.6.

Top-k JTTs Algorithm

Our *Top-k JTTs Algorithm* combines four metrics in determining the top-k results for a query q, namely, preferential dominance, degree of relevance, coverage and diversity. To compute the overall result, we use a number of heuristics that guide the order of generation of the JTTs. We first evaluate the performance of our basic heuristics and then show their effectiveness.

First, we evaluate the performance of our underlying diversification heuristic by comparing it against the brute-force algorithm both in terms of the quality of produced results as well as the time. The complexity of all methods depends on the number m of candidate trees to choose from and on the required number k of trees to select. We experiment with a number of different values for m and k. However, the exponential complexity of the brute-force algorithm prevents us from using large values for these two parameters. Therefore, we limit our study to $m = 10, 20, 30$ and $k = 4, 8, 12, 16, 20$. In Table 5.2,

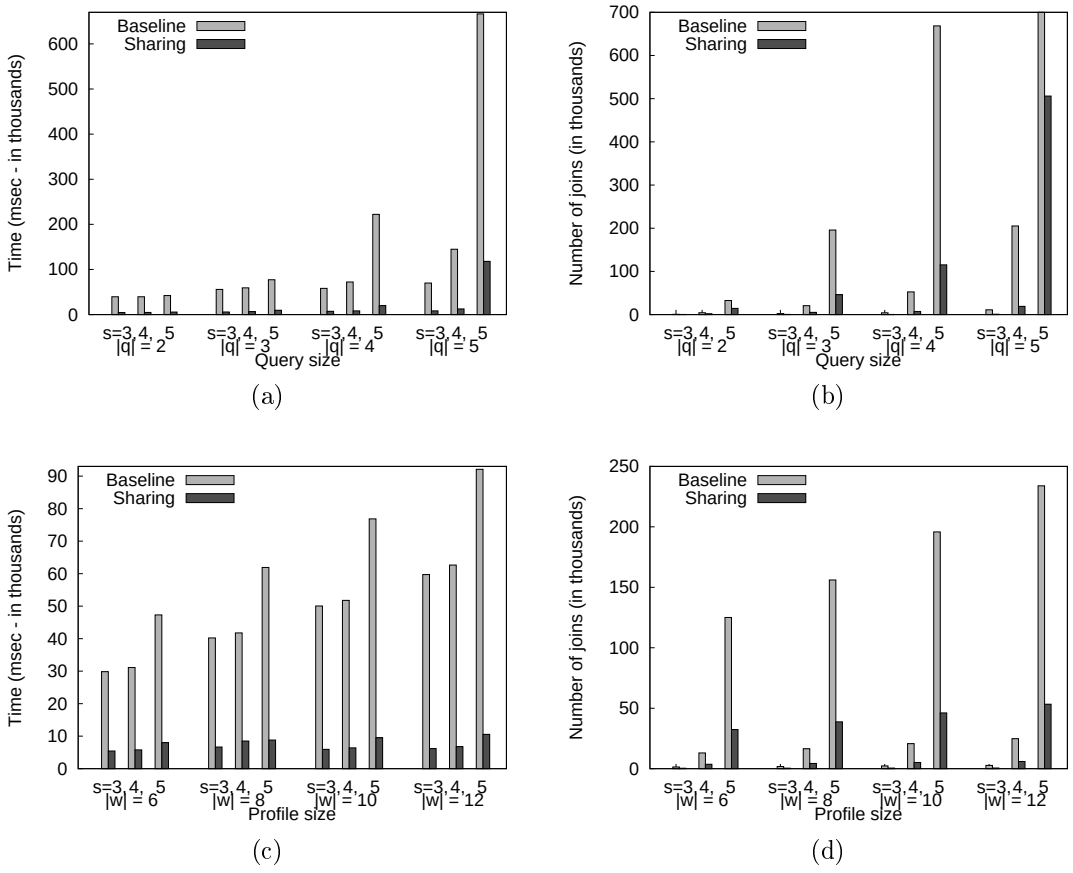


Figure 5.6: Movie dataset: Total time for (a) a fixed profile and (c) a fixed query and total number of join operations for (b) a fixed profile and (d) a fixed query.

we show the results for the brute-force method and our heuristic. We observe that the brute-force method consumes much more time than the heuristic, while the set diversity of the produced results is similar (the difference is less than 1%). Applying our diversity heuristic improves the set-diversity of results, even when choosing trees of tuples only among those computed for the choice keywords of the same winnow level (Figure 5.7).

As discussed, we can tune the trade-off between dominance and relevance through the function $\mathcal{F}$. To demonstrate this, we use the function $\mathcal{F}(i) = k \cdot \frac{L-(i-1)}{\sum_i (L-(i-1))}$ for various values of L, where L is the lowest winnow level from which results are retrieved. For example, when $L = 1$, only results corresponding to the choice keywords of the first winnow level are returned. We run the following experiments for $|q| = 1$, $|w| = 10$ and $s = 4$. In Figures 5.8a, 5.8b, we use a profile leading to five winnow levels. We also consider a sixth level containing the query results when no preferences are used. We show the average normalized dominance and relevance respectively, for $L = 1, 2, \ldots, 6$. Given a set S of JTTs, the average dominance is the mean *dorder* and *iorder* of the trees in $PRes(Q, P)$ and $Res(Q)$ respectively, where for those JTTs in $Res(Q)$ that are not the projection of any JTT in $PRes(Q, P)$, we use $iorder = 6$ (that is, the maximum winnow level plus 1) as opposed to ∞. As L increases, the average dominance decreases because

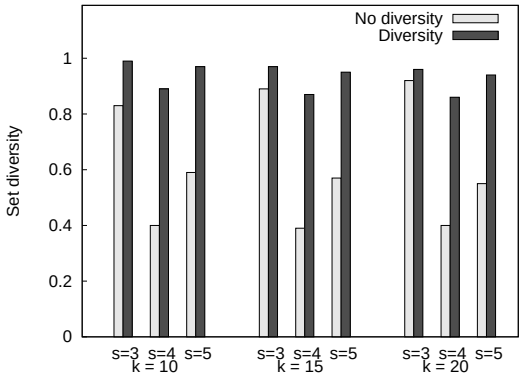


Figure 5.7: Movie dataset: Set diversity of first-level results.

Table 5.2: Brute-force vs. Heuristic diversification.

m	k	Brute-force		Heuristic	
		Set diversity	Time	Set diversity	Time
10	4	0.98	33	0.97	7
	8	0.92	38	0.92	11
20	4	0.99	623	0.97	16
	8	0.94	71194	0.93	21
	12	0.86	171315	0.86	30
	16	0.81	11730	0.80	43
30	4	1.00	3190	0.99	21
	8	0.98	3041457	0.98	30
	12	0.96	105035021	0.95	43
	16	0.94	300561487	0.93	61
	20	0.91	104214544	0.90	79

less preferable choice keywords are also employed, while the average relevance increases, since highly relevant JTTs from the lower levels enter the top-k results.

Coverage is also very important, especially in the case of skewed selectivity among the choice keywords. For example, if the combination of the query keywords and some top-level choice keyword is very popular, then, without coverage, the JTTs computed for that choice keyword would dominate the result. Figure 5.8c shows the average coverage for two profiles, when our coverage heuristic is employed or not, for $L = 5$. The first profile (*Pr. A*) contains keywords with similar selectivities, while the second one (*Pr. B*) contains keywords of different popularity, i.e. some keywords produce more results than others. Coverage is greatly improved in both cases when the heuristic is applied, since more keywords from all winnow levels contribute to the result. The improvement is more evident for *Pr. B*, as expected.

In general, high coverage ensures that results reaching the users represent most of their

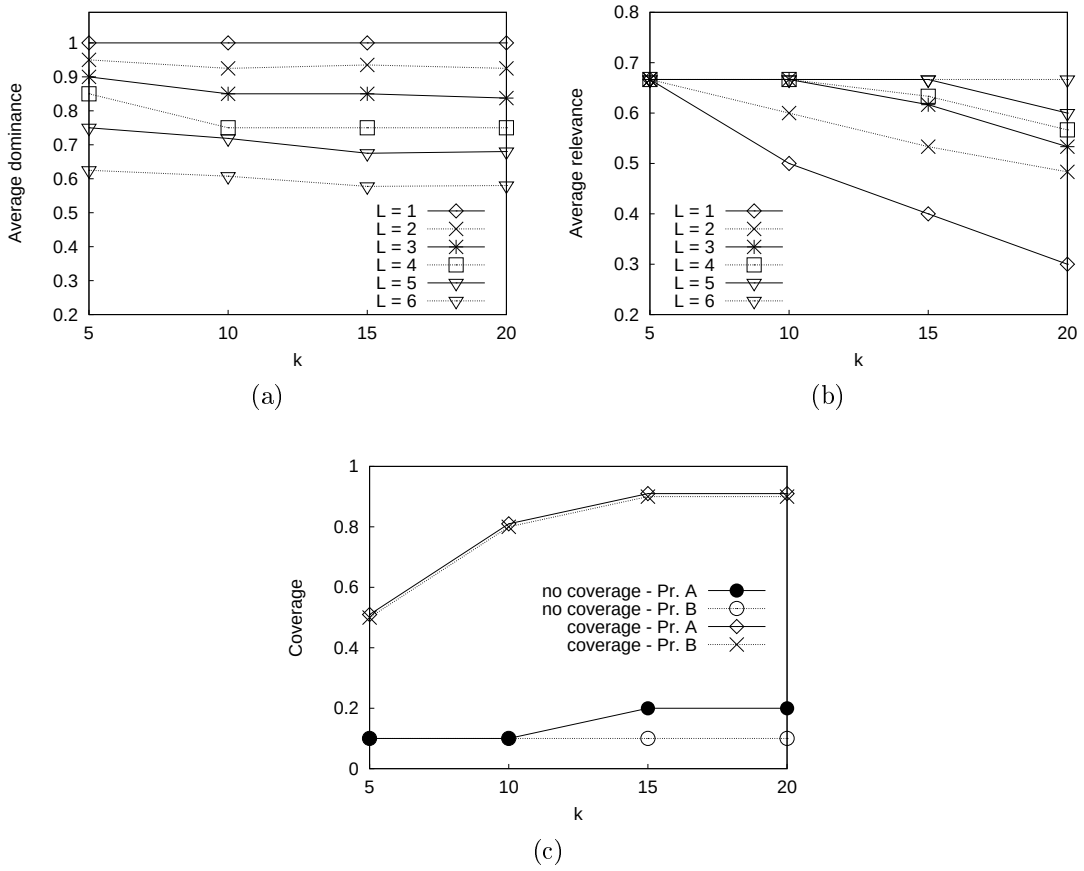


Figure 5.8: (a) Average dominance, (b) average relevance and (c) coverage.

interests. However, this does not necessarily mean that those results are not similar with each other. Next, to demonstrate how selecting results based on our diversity heuristic can produce even more satisfying results, we execute a number of queries and present here a characteristic example. For the query $\{drama\}$ and a winnow level containing the keywords *Greek* and *Italian*, four results should be selected according to $\mathcal{F}$. For simplicity, we use only the JTTs computed for the joining trees of tuple sets $Movies^{\{Drama\}} - Directors^{\{Greek\}}$ and $Movies^{\{Drama\}} - Directors^{\{Italian\}}$. When only coverage is applied, the results are:

(i) (Tan21, Eternity and a Day, 1998, Th. Angelopoulos, **Drama**) – (Th. Angelopoulos, 1935, **Greek**)

(ii) (FF50, Intervista, 1992, F. Fellini, **Drama**) – (F. Fellini, 1920, **Italian**)

(iii) (Tan12, Landscape in Fog, 1988, Th. Angelopoulos, **Drama**) – (Th. Angelopoulos, 1935, **Greek**)

(iv) (GT01, Cinema Paradiso, 1989, G. Tornatore, **Drama**) – (G. Tornatore, 1955, **Italian**)

When diversity is also considered, the third of these results is replaced by (PvG02, Brides, 2004, P. Voulgaris, **Drama**) – (P. Voulgaris, 1940, **Greek**). Coverage remains the same, however, with diversity, one more director can be found in the results.

Result Pruning and Time Overhead

Finally, we study the overall impact of query personalization in keyword search in terms of the number of returned results and the corresponding time overhead. Both of these measures depend on how frequently the relative to the query choice keywords appear in the database. Therefore, we experiment with profiles with different selectivities. As *profile selectivity* we define the normalized sum of the number of appearances of each choice keyword in the database. We use profiles with $|w| = 6, 8, 10, 12$ and study a query with $|q| = 2$: (i) when no preferences are applied or a profile with (ii) small and (iii) large selectivity is used.

In Figures 5.9a and 5.9b, we measure the total number of the constructed JTTs, i.e. not just the top-k ones, for $s = 3, 4$ respectively. In general, query personalization results in high pruning. For $s = 3$, the use of the profile with large selectivity prunes more than 85% of the initial results, while for the profile with small selectivity the pruning is more than 95%. The respective percentages for $s = 4$ are 33% and 74%. In Figures 5.9c and 5.9d, we measure the time to generate the joining trees of tuple sets required to retrieve the final results. When the profile with large selectivity is applied, the time overhead is 24% for $s = 3$ and 35% for $s = 4$ on average. For the profile with small selectivity, the corresponding percentages are 22% and 32% on average.

5.5.2 Usability Evaluation

The goal of our usability study is to demonstrate the effectiveness of using preferences. In particular, the objective is to show that for a reasonable effort of specifying preferences, users get more satisfying results. To this end, we conducted an empirical evaluation of our approach using the Movie dataset, with 10 computer science students with a moderate interest in movies. Each of them provided a set of contextual keyword preferences including context-free ones. On average, there were five preferences related to each of the queries that were later submitted by each user. Users were asked to evaluate the quality of the top-10 JTTs retrieved. For characterizing the quality, we use two measures: (i) precision and (ii) degree of satisfaction. The first one captures the judgment of the users for each individual result. In particular, users marked each result with 1 or 0, indicating whether they considered that it should belong to the top-10 ones or not. The ratio of 1s corresponds to the precision of the top-10 results, namely $precision(10)$. The second measure evaluates the perceived user satisfaction by the set of results as a whole. To assess this, users were asked to provide an overall degree of satisfaction (dos) in the range [1, 10] to indicate how interesting the overall result set seemed to them.

We compare the results of keyword queries when executing them: without using any of the preferences and with using the related contextual keyword preferences, first based only on dominance and relevance and then based on all four properties. Also, we consider using only context-free preferences as well as a case in which there is no preference with context equal to the query and so, relaxation is employed. We use $\mathcal{F}$ as in our perfor-

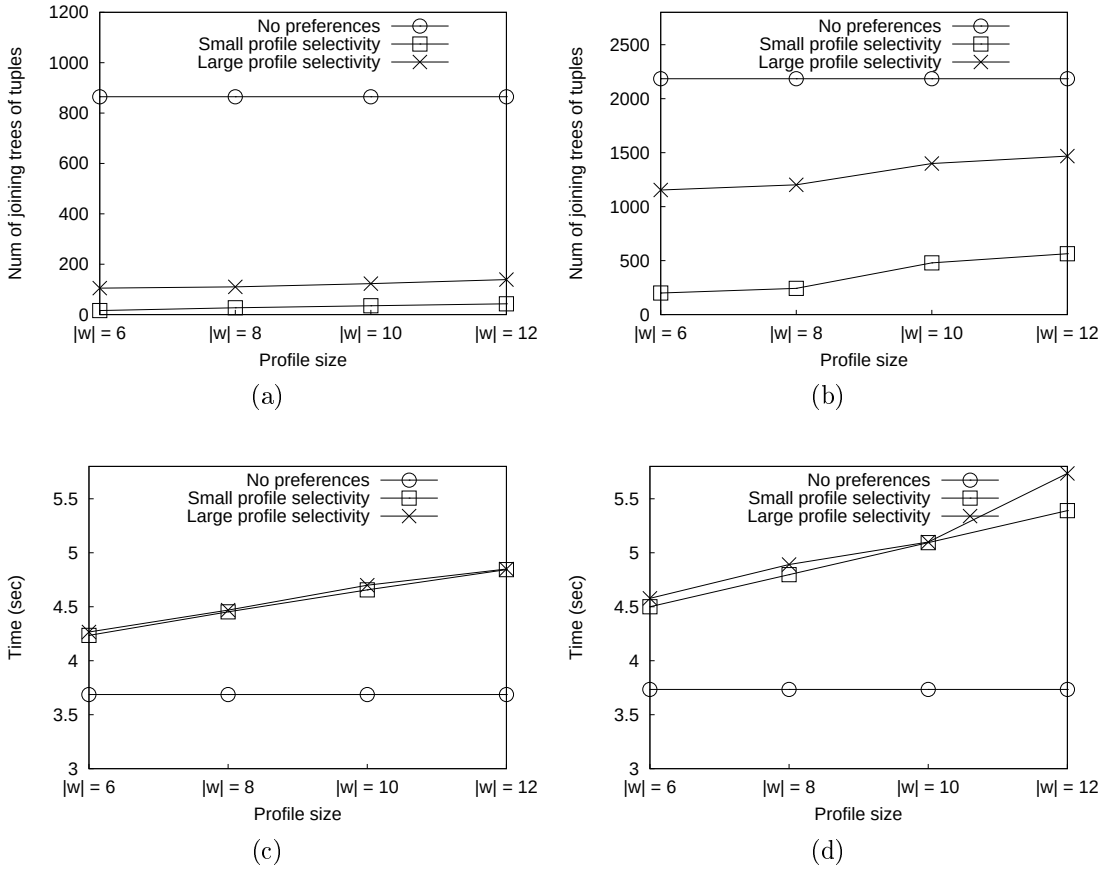


Figure 5.9: Number of joining trees of tuples for (a) $s = 3$ and (b) $s = 4$ and time overhead for (c) $s = 3$ and (d) $s = 4$.

mance experiments with L equal to the maximum winnow level for each user. Table 5.3 reports the average values of the quality measures (we omit the detailed per user scores due to space limitations). Our results indicate that, when no preferences are employed, both *precision* and *dos* are low. The use of context-free preferences improves both measures moderately, since such preferences capture only the generic interests of each user. Applying contextual keyword preferences improves quality considerably, even when preferences with relaxed context are employed. The most satisfying results are produced when all properties are taken into account. This demonstrates how important set-centric properties are when combined with dominance and relevance. Although our evaluation is preliminary, we believe that the results attained so far are promising.

Concerning user behavior, in general, most of our users defined preferences that resulted in short graphs of choices. Short graphs produce few winnow levels and consequently, many ties among the results with respect to preferential dominance. Using relevance, coverage and diversity led to resolving such ties. We also noticed that our users were often positively biased for movies they have heard about. This can be seen as an indication that exploiting previous queries to generate additional preferences based on popularity (as explained in Section 5.3) can prove very useful.

Table 5.3: Usability Evaluation.

	precision(10)	dos
No Preferences	0.09	1.9
Context-Free Keyword Preferences	0.21	2.7
Relaxed Context	0.87	7.4
Contextual Keyword Preferences		
Dominance-Relevance	0.89	7.9
Dominance-Relevance-Coverage-Diversity	0.94	8.7

5.6 Summary

The simplicity of keyword-based queries makes them a very popular method for searching. However, keyword-based search may return a large amount of matching data, often loosely related to the actual user intent. In this chapter, we have proposed personalizing keyword database search by employing preferences. By extending query-relevance ranking with preferential ranking, users are expected to receive results that are more interesting to them. To further increase the quality of results, we have also suggested selecting k representative results that cover many user interests and exhibit small overlap. We have presented algorithms that extend current schema-based approaches for keyword search in relational databases to incorporate preference-based ranking and top-k representative selection.

The results presented in this chapter also appear in [106].

Chapter 6

Preference-Aware Publish/Subscribe Delivery

With the explosion of the amount of information that is currently available on-line, publish/subscribe systems offer an attractive alternative to searching by providing a proactive model of information supply. In such systems, users express their interest in specific pieces of data (or *events*) via *subscriptions*. Then, they are *notified* whenever some other user generates (or *publishes*) an event that *matches* one of their subscriptions. Typically, all subscriptions are considered equally important and users are notified whenever a published event matches any of their subscriptions.

However, getting notified about all matching events may lead to overwhelming the users with huge amounts of notifications, thus hurting the acceptability of publish/subscribe systems. To control the rate of notifications received by the subscribers, it would be useful to allow them to rank the importance or relevance of events. Then, they would only receive notifications for the most important or relevant among them. For example, take a user Addison that generally likes drama movies but prefers drama movies directed by

T. Burton to drama movies directed by S. Spielberg. Ideally, Addison would like to receive notifications about S. Spielberg drama movies only if there are no, or not enough, notifications about T. Burton drama movies.

In this chapter, we propose extending subscriptions to allow users express the fact that some events are more important or relevant to them than others. To indicate priorities among subscriptions, we introduce *preferential subscriptions*. We show how to formulate preferences among subscriptions using the qualitative and the quantitative approach. Events are ranked so that an event that matches a highly preferred subscription is ranked higher than an event that matches a subscription with a lower preference.

Based on preferential subscriptions, we introduce a top-k variation of the publish/subscribe paradigm in which users receive only the matching events having the k highest ranks as opposed to all events matching their subscriptions. Since the generation of events is continuous, we also introduce a number of delivering policies that determine the range of events over which the top-k computation is performed.

However, the top-k events are often very similar to each other. Besides pure accuracy achieved by matching the criteria set by the users, diversification, i.e. recommending items that differ from each other, has been shown to increase user satisfaction [136]. For instance, our user Addison would probably like to receive information about different drama movies by T. Burton as well as a couple of S. Spielberg's drama movies once in a while. To this end, we adjust the top-k computation to take also into account the *diversity* of the delivered events. To achieve this, we consider both the importance of each event as specified by the user preferences as well as its diversity from other top-ranked events.

In a nutshell, in this chapter, we

- propose personalizing publish/subscribe delivery through preferences among users subscriptions,
- introduce a top-k variation of the publish/subscribe paradigm,
- adjust the top-k computation to take into consideration the diversity of the delivered events and
- study a number of delivering policies for forwarding events.

We have implemented a prototype, termed PrefSIENA [5]. PrefSIENA extends SIENA [6], a popular publish/subscribe middleware system, with preferential subscriptions, delivering policies and diversity towards achieving top-k event delivery. We present a number of experimental results to assess the number of events delivered by PrefSIENA with respect to the original SIENA system, as well as their rank and diversity. We also report on the overheads of supporting diversity-aware top-k delivery.

The rest of the chapter is structured as follows. Section 6.1 presents publish/subscribe preliminaries. Section 6.2 introduces preferential subscriptions and event ranks. In Section 6.3, we focus on how to diversify the top-ranked events, while in Section 6.4, we examine a number of different delivering policies for forwarding events. In Section 6.5,

we introduce an algorithm for computing the top-ranked events based on preferential subscriptions and in Section 6.6, we present our evaluation results. Section 6.7 describes related work and finally, Section 6.8 concludes the paper.

6.1 Publish/Subscribe Preliminaries

In general, a publish/subscribe system consists of three parts: (i) the publishers that provide events to the system, (ii) the subscribers that enter subscriptions and consume events and (iii) an event-notification service that stores the various subscriptions, matches the incoming events against them and delivers the matching events to the appropriate subscribers ([45]). Publishers can publish events at any time and these events will be delivered to all interested subscribers at some point in the future.

We use a generic way to form events, similar to the one used in [26, 46]. In particular, events are sets of typed attributes. Each event consists of an arbitrary number of attributes and each attribute has a type, a name and a value. Attribute types belong to a predefined set of primitive types, such as "integer" or "string". Attribute names are character strings that take values according to their type. An example event about a movie is shown in Figure 6.1a. Formally:

An *event* e is a set of typed attributes $\{a_1, \ldots, a_p\}$, where each a_i, $1 \leq i \leq p$, is of the form $(a_i.type\ a_i.name = a_i.value)$.

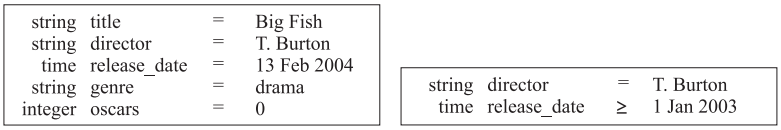
Subscriptions are used to specify the kind of events users are interested in. Each subscription consists of a set of constraints on the values of specific attributes. Each attribute constraint has a type, a name, a binary operator and a value. Types, names and values have the same form as in events. Binary operators include common operators, such as, $=$, $\neq$, $<$, $>$ and *substring*. An example subscription is depicted in Figure 6.1b. Formally:

A *subscription* s is a set of attribute constraints $\{b_1, \ldots, b_q\}$, where each b_i, $1 \leq i \leq q$, is of the form $(b_i.type\ b_i.name\ \theta_{b_i}\ b_i.value)$, $\theta_{b_i} \in \{=, <, >, \leq, \geq, \neq, substring, prefix, suffix\}$.

Intuitively, we can say that an event e *matches* a subscription s, or alternatively s *covers* e, if and only if, every attribute constraint of s is satisfied by some attribute of e. Formally:

Definition 6.1 (Cover Relation)**.** Given an event $e = \{a_1, \ldots, a_p\}$ and a subscription $s = \{b_1, \ldots, b_q\}$, s covers e ($s \succ e$), if and only if, $\forall\ b_j \in s$, $\exists\ a_i \in e$, such that, $a_i.type = b_j.type$, $a_i.name = b_j.name$ and $((a_i.value)\ \theta_{b_j}\ (b_j.value))$ holds, $1 \leq i \leq p$, $1 \leq j \leq q$.

An event e is delivered to a user, if and only if, the user has submitted at least one subscription s, such that s covers e. For example, the subscription of Figure 6.1b covers



(a) (b)

Figure 6.1: (a) Event and (b) subscription examples.

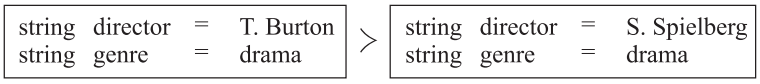


Figure 6.2: Qualitative preference example.

the event of Figure 6.1a, and therefore, this event will be delivered to all users who have submitted this subscription.

6.2 Preference Model

In this section, we first extend subscriptions to include preferences. Then, we examine how to compute the importance of published events for the users.

6.2.1 Preferential Subscriptions

Our goal is for each subscriber, instead of receiving all matching events, to receive only the most interesting among them. To achieve this, we allow users to express preferences along with their subscriptions. In general, preferences can be expressed using either a quantitative or a qualitative approach. Following a *quantitative* preference model, users explicitly provide numeric scores to indicate their degree of interest (e.g. [11, 80, 113]). Following a *qualitative* model, users employ binary relations to directly define preferences between data items (e.g. [32, 51, 74]). We first use a qualitative preference model [32], since this model is more general than the quantitative one and also closer to the user's intuition. Specifically:

Definition 6.2 (Preferential Subscription Model)**.** Let S^X be the set of subscriptions of user X. Along with S^X, X specifies a binary preference relation C^X on S^X, $C^X = \{(s_i \succ s_j) \mid s_i, s_j \in S^X\}$, where $s_i \succ s_j$ denotes that X prefers s_i over s_j or considers s_i more interesting than s_j.

An example is shown in Figure 6.2. Given C^X, we would like to rank subscriptions based on interest. To this end, we use the winnow operator [32]. The intuition is to assign the highest rank to the most preferred subscriptions, that is, to those subscriptions for which there is no other subscription in S^X that is preferable over them. Formally, winnow at level 1, $win^X(1)$, is the set of subscriptions $s_i \in S^X$ for which, $\nexists\ s_j \in S^X$ with $(s_j \succ s_i) \in C^X$. An additional application of winnow, $win^X(2)$, returns the next

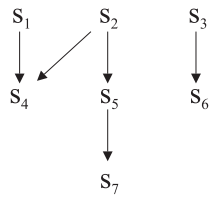
Preference Relation	Graph	Preference Ranks
$s_1 \succ s_4$		s_1, s_2, s_3: 1
$s_2 \succ s_4$		s_4, s_5, s_6: 2/3
$s_2 \succ s_5$		s_7: 1/3
$s_3 \succ s_6$		
$s_5 \succ s_7$		

Figure 6.3: Extracting preference ranks.

most preferred subscriptions, that is, $s_i \in win^X(2)$, if and only if, $\nexists\ s_j \in (S^X - win^X(1))$ with $(s_j \succ s_i) \in C^X$. Generalizing, the winnow operator at level l, $l > 1$, returns a set of subscriptions, $win^X(l)$, consisting of the subscriptions $s_i \in (S^X - \cup_{q=1}^{l-1} win^X(q))$ such that $\forall\ s_i \in win^X(l)$, $\nexists s_j \in (S^X - \cup_{q=1}^{l-1} win^X(q))$ with $(s_j \succ s_i) \in C^X$. Repeated applications of winnow result in ranking all subscriptions in S^X.

The straightforward way to compute win is to iterate through all subscriptions in S^X. Instead, if the preference relation is acyclic, we can organize subscriptions in a directed preference graph, where there is one node in the graph for each subscription in S^X and an edge from a node representing subscription s_i to a node representing subscription s_j, if and only if, $(s_i \succ s_j) \in C^X$. Now, a topological sort of this graph can be used to compute win. In the first iteration, we output all nodes with no incoming edges. These nodes correspond to all subscriptions $s_i \in win^X(1)$, since for these, there is no $s_j \in S^X$ with $(s_j \succ s_i) \in C^X$. In the next iteration of the algorithm, these nodes are removed from the graph, along with their outgoing edges, and the nodes without incoming edges are output. Clearly, these nodes correspond to all subscriptions $s_i \in win^X(2)$. The algorithm stops when all nodes in the preference graph have been processed.

We associate a preference rank, $prefrank_i^X$ with each subscription s_i based on the winnow level that the subscription belongs to. Since subscriptions retrieved earlier are of higher interest to the users, subscriptions returned at level l of the winnow operator are assigned a preference rank equal to $\mathcal{G}(l)$, where $\mathcal{G}$ is a strictly monotonically decreasing function for which $\mathcal{G}(l) \mapsto [0, 1]$. Thus, for each user, we get pairs of subscriptions and preference ranks.

Definition 6.3 (Preferential Subscription)**.** A preferential subscription ps_i^X of user X is a pair of the form $ps_i^X = (s_i, prefrank_i^X)$, where s_i is a subscription and $prefrank_i^X$ is a real number in $[0, 1]$ that expresses the degree of interest of X for s_i.

For instance, Figure 6.3 depicts the preference graph for an example preference relation and the extracted preference ranks when $\mathcal{G}(l) = (D + 1 - (l - 1))/(D + 1)$, where D is the diameter of the preference graph. The cover relation (Definition 6.1) is extended to preferential subscriptions as follows: Given an event e and a preferential subscription $ps_i^X = (s_i, prefrank_i^X)$, $ps_i^X \succ e$, if and only if, $s_i \succ e$.

Alternatively, instead of providing C^X, users could explicitly provide an interest score $prefrank_i^X$ for each of their subscriptions. This would correspond to a quantitative ap-

string	director	=	T. Burton	0.8
string	genre	=	drama	

string	director	=	S. Spielberg	0.6
string	genre	=	drama	

Figure 6.4: Quantitative preferences examples.

proach. A higher preference rank indicates a more important subscription. Examples are shown in Figure 6.4.

6.2.2 Computing Event Ranks

Let P^X be the set of preferential subscriptions of user X. We use these preferential subscriptions to rank the published events and deliver to the user only the highest ranked ones. We define the *rank* of an event to be a function $\mathcal{F}$ of the preference ranks of the subscriptions that cover it.

Instead of using the preference ranks of all covering subscriptions, we use only the preference ranks of the *most specific* ones. A subscription s is a most specific one if no other subscription in P^X is covered by it, where:

Definition 6.4 (Cover between Subscriptions)**.** Given two subscriptions s_i and s_j, s_i covers s_j, if and only if, for each event e such that $s_j \succ e$, it holds that $s_i \succ e$.

For example, assume the event of Figure 6.1a and the preferential subscriptions ($\{genre = drama\}$, 0.7) and ($\{genre = drama, director = T.\ Burton\}$, 0.9) by Addison and ($\{genre = drama\}$, 0.7) and ($\{genre = drama, director = T.\ Burton\}$, 0.5) by another user Carson (for ease of presentation, we omit the type of each attribute). Both subscriptions of each user cover the event. Between the two, for each user, the latter subscription is more specific than the former one, in the sense that in the latter subscription the user imposes an additional, more specific requirement to movies (Addison prefers T. Burton's dramas over the rest, while Carson thinks that T. Burton's dramas are worse than other dramas). Thus, intuitively, the preference rank of the latter subscription should superimpose that of the former one, whenever an event matches both of them.

The event rank is formally defined as follows:

Definition 6.5 (Event Rank)**.** Given an event e, a user X, the set P^X of the user's preferential subscriptions and the set $P_e^X = \{(s_1, prefrank_1^X), \ldots, (s_m, prefrank_m^X)\}$, $P_e^X \subseteq P^X$, such that $s_i \succ e$, $1 \leq i \leq m$, of the most specific subscriptions that cover e, the event rank of e for X is equal to $rank(e, X) = \mathcal{F}(prefrank_1^X, \ldots, prefrank_m^X)$, where $\mathcal{F}$ is a monotonically increasing function.

User X prefers an event e_i over the event e_j, if and only if, $rank(e_i, X) > rank(e_j, X)$. As the aggregation function $\mathcal{F}$ for computing the rank of an event, we may use the

Ranking of events: $e_1(comedy), e_2(drama), e_3(drama), e_4(drama), e_5(horror), e_6(sci\text{-}fi)$						
Top-4 events based on their ranks: e_1, e_2, e_3, e_4						
Diverse Top-4 events:						
$divrank(e_1, X)$	$divrank(e_2, X)$	$divrank(e_3, X)$	$divrank(e_4, X)$	$divrank(e_5, X)$	$divrank(e_5, X)$	$L^X = \emptyset$
-	-	-	-	-	-	$L^X = (e_1, e_4)$
-	-	0.4	0.4	0.85	0.8	$L^X = (e_1, e_4, e_5)$
-	-	0.4	0.4	-	0.8	$L^X = (e_1, e_4, e_5, e_6)$

Figure 6.5: Computing top-4 diverse events.

maximum, mean, minimum or a weighted sum of the preference ranks of its covering subscriptions.

Now, we can formally define preferential top-k delivery:

Definition 6.6 (Preferential Top-k Delivery)**.** Given a set M of n matching events for a user X, deliver a subset L, $L \subseteq M$, with cardinality k, such that, $rank(e_i, X) \geq rank(e_j, X)$, $\forall\ e_i \in L$, $e_j \in M \backslash L$.

6.3 Event Diversity

Many times, the events that eventually reach the user are very similar to each other. However, it is often desirable that these events exhibit some diversity. In this section, we examine how to reduce the similarity of the matching events forwarded to the users. First, we introduce diversity-aware delivery and then describe how to integrate preferences and diversity towards improving the information quality of the delivered events.

6.3.1 Diversity-Aware Matching

Instead of overwhelming users with matching events that are all very similar to each other, we opt to select a representative set of events according to their diversity. To measure the *diversity* of events, i.e. how different they are, we first define the distance between two events. Without loss of generality, we assume that the events have the same number of attributes. Otherwise, we can simply append a sufficient number of "dummy" attributes to the event having the smaller number of attributes.

Definition 6.7 (Event Distance)**.** Given two events $e_1 = \{a_1, \ldots, a_p\}$ and $e_2 = \{a'_1, \ldots, a'_p\}$, the distance between e_1 and e_2 is defined as:

$$d(e_1, e_2) = 1 - \frac{\sum_{i=1}^{p} \delta_i w_i}{\sum_{i=1}^{p} w_i},\ \text{where}\ \delta_i = \begin{cases} 1 & \text{if } a_i = a'_i \\ 0 & \text{otherwise} \end{cases}$$

and each w_i is an attribute specific weight, $1 \leq i \leq p$.

Based on the above definition, the distance of any two events decreases as the number of their common attributes increases. Weights express the importance of an attribute for

a specific application or user. In lack of such application-dependent information, we can assign equal weights to all attributes.

A number of different definitions of set diversity have been proposed in the context of recommender systems; here we model diversity as the aggregate or, equivalently, average distance of all pairs of events in the set [134]. We use the term "set" loosely to denote a set with *bag semantics* or a *multi-set*, where the same event may appear more than once in the set.

Definition 6.8 (Set Diversity)**.** Given a set of m events $L = \{e_1, \ldots, e_m\}$, the set diversity of L is:

$$div(L) = \frac{\sum_{i=1}^{m} \sum_{j>i}^{m} d(e_i, e_j)}{(m-1)m/2}.$$

Now, the diversity-aware delivery problem can be defined as follows:

Definition 6.9 (Diverse Top-k Delivery)**.** Given a set M of n matching events, $|M| = n$, deliver a subset L, $L \subseteq M$, with cardinality k, such that,

$$div(L) = \max_{L' \subseteq M, |L'|=k} \{div(L')\}.$$

The problem of selecting the k items having the maximum average pair-wise distance out of n items is similar to the *p-dispersion-sum* problem. This problem as well as other variations of the general p-dispersion problem (i.e. select p out of n points so that the minimum distance between any two pairs is maximized) have been extensively studied in operations research and are in general known to be NP-hard [42, 43].

A brute-force method to identify the k most diverse events in M is to first produce all $\binom{n}{k}$ possible combinations of k events, and then pick the one with the maximum set diversity. The complexity of this process in terms of the required event distance computations is equal to $\frac{n!}{k! \cdot (n-k)!} \cdot \frac{n \cdot (n-1)}{2}$ and therefore, the computational cost is too high even for relatively small values of n and k.

Instead, we use the following intuitive heuristic. We incrementally construct a diverse subset of events by selecting at each step an event e that is furthest apart from the set of events already selected. The distance of an event e from a set of events $L = \{e_1, \ldots, e_m\}$ is defined as:

$$dis(e, L) = \min_{1 \leq i \leq m} d(e, e_i).$$

In particular, let $M = \{e_1, \ldots, e_n\}$ be the input set of n matching events and L be the set we want to construct. Initially, L is empty. We first add to L the two furthest apart elements of M. Then, we compute the distances $dis(e_i, L)$, $\forall e_i$, such that $e_i \in M \backslash L$ and add to L the event with the maximum corresponding distance. This process is repeated until k events have been added to L. With this method, the required number of event distance operations are equal to $\frac{n \cdot (n-1)}{2} + [2 \cdot (n-2) + \ldots + (k-1) \cdot (n-k+1)]$. We can further reduce the number of performed operations based on the observation that after the insertion of an event e to L, the distances of all other events that have not yet entered the diverse events from L', $L' = L \cup \{e\}$, are affected only by the presence of e.

Property 6.1. *Given an event e and two sets L and $L' = L \cup \{e\}$, the distance of an event e' from L' is:*

$$dis(e', L') = \min\{dis(e', L), d(e', e)\}.$$

Using Property 6.1, the required event distance operations are equal to $\frac{n\cdot(n-1)}{2} + 2 \cdot (n-2) + (n-3) + \ldots + (n-k+1)$. Algorithm 17 summarizes the above procedure.

Diverse Events Algorithm

Input: A set M of matching events for user X.

Output: A subset L of k diverse events.

```
1: begin
2: L ← ∅;
3: find the events e1, e2 ∈ M s.t.
   d(e1, e2) = max{d(ei, ej)|ei, ej ∈ M, i ≠ j};
4: L ← L ∪ {e1, e2};
5: for all ei ∈ M\L do
6:    disi ← dis(ei, L);
7: find the event eadd s.t. disadd = max{disi|ei ∈ M\L};
8: L ← L ∪ {eadd};
9: while |L| < k do
10:   for all ei ∈ M\L do
11:      disi ← min{disi, d(ei, eadd)};
12:   find the event eadd s.t. disadd = max{disi|ei ∈ M\L};
13:   L ← L ∪ {eadd};
14: return L;
15: end
```

Algorithm 17: Diverse Events Algorithm

6.3.2 Diverse Top-k Preference Ranking

We would like to combine both diversity and preference ranking when selecting which events to forward, so that the delivered events are both highly preferred as well as diverse with each other, i.e. we want to select k out of n events so that both the average of their preference ranks and their diversity are as good as possible. To this end, we combine the two measures to produce a combined ranking:

Definition 6.10 (Diversity-Aware Set Rank)**.** Let X be a user. Given a set of m events $L = \{e_1, \ldots, e_m\}$, the diversity-aware rank of L for X is

$$divrank(L, X) = \sigma \cdot \frac{\sum_{i=1}^{m} rank(e_i, X)}{m} + (1 - \sigma) \cdot div(L).$$

where $\sigma \in [0, 1]$. When $\sigma = 0$ (resp. $\sigma = 1$), events are chosen based only on diversity (resp. preference rank).

Now the problem becomes:

Definition 6.11. (TOP-k PREFERRED DIVERSITY-AWARE DELIVERY) Given a set M of n matching events for a user X, deliver a subset L, $L \subseteq M$, with cardinality k, such that,

$$divrank(L, X) = \max_{L' \subseteq M, |L'|=k} \{divrank(L', X)\}.$$

To locate the k events with the maximum $divrank$, we use Algorithm 17, where we replace the $d(e_1, e_2)$ and dis_i functions with the corresponding $divrank$ versions.

In the following example, we apply Algorithm 17 to six events e_1, e_2 ..., e_6. To simplify our example, we assume that all events have only the attribute *genre* with value equal to *comedy*, *drama*, *drama*, *drama*, *horror*, *sci-fi* and event ranks 0.9, 0.8, 0.8, 0.8, 0.7, 0.6 respectively for a given user X. The distance between two events with the same genre is 0, while the distance between two events with different genres is 1. Figure 6.5 shows the trace of the heuristic applied on our example when $k = 4$ and $\sigma = 0.5$. We resolve ties in the case of events with the same $divrank$ values by selecting the most recently published events.

6.4 Delivery Modes

Publish/subscribe systems offer an asynchronous mode of communication between publishers and subscribers by decoupling event publication from event delivery. In general, each event e is associated with a number of time instants:

1. The time e is published ($tpub_e$)
2. The time e reaches the event-notification service ($tserv_e$)
3. The time e is matched against subscriptions ($tmatch_e$)
4. The time e is forwarded to the user ($tforw_e$) and
5. The time e is actually received by the user ($trecv_e$)

Since events are continuously published and matched, we need to define over which sets of this stream of matching events we apply preference ranking and diversification. In the following, we use $tpub_e$ as the time instant associated with each event, since this is the time that characterizes best its freshness. However, note that, in general, events may reach the event-notification service and be matched in an order different from their publication order. Although out-of-order delivery does not invalidate our definitions, it may, however, complicate their implementation. Note that, alternatively, one could replace $tpub_e$ with $tmatch_e$ in all our definitions. This makes their implementation easier, but complicates semantics especially in the case of a distributed notification service.

We consider three fundamental modes of forwarding events, namely: (i) periodic, (ii) sliding-window and (iii) history-based filtering delivery. With *periodic delivery*, the top-k

events are computed over disjoint periods of length T and forwarded to the subscribers once at the end of each period. With *sliding-window* delivery, the top-k events are computed over sliding windows of length w, so that an event is forwarded, if and only if, it belongs to the top-k events in the current window. Finally, *history-based filtering* continuously forwards new events as they are matched, if and only if, they are better than the top-k events recently delivered.

The lengths T and w can be defined either in time units (e.g. as "the top-10 events matched per hour" and respectively, "the top-10 events matched in the last hour") or in number of events (e.g. as "the top-10 events per 100 matched ones" and respectively, "the top-10 events among the 100 most recently matched ones"). For clarity, in the following, we define T in terms of time units and w in terms of events, since this seems to fit better with the corresponding delivery modes. Next, we describe the three delivery modes in detail.

6.4.1 Periodic Delivery

Periodic delivery is appropriate for subscribers who wish to receive a list of important events regularly, for example, every morning when they reach their office or once in an hour. In this case, time is divided into disjoint periods of duration T and top-ranked events are computed within each period. Whenever a period ends, the k highest ranked matching events published within this period are forwarded to the users. To improve the freshness of events, ties are resolved by choosing to forward the most recent among the tied events. Formally:

Definition 6.12 (Periodic Top-k)**.** Let X be a user and M be the set of matching events published during a period starting at time instant t, i.e. an event $e_i \in M$, if and only if, $t \leq tpub_{e_i} < t + T$. Let L be a subset of M with k events that has the maximum $divrank(L, X)$ among all subsets of M with the same cardinality. If there are more than one such subsets, let L' be one with the maximum $\sum_{e_i \in L'} tpub_{e_i}$ among them. If again, there are more such sets, we randomly select one of them, say L_P. An event e with publication time $tpub_e$, $t \leq tpub_e < t + T$, is forwarded, if and only if, $e \in L_P$.

The number of events forwarded using periodic delivery is fixed and depends only on k and T. Thus, we achieve a constant event delivery rate of $k \cdot \lfloor c/T \rfloor$ events in every time interval of duration c.

As an example, assume a single user, say Addison, who is interested in receiving events about movies showing in theaters. Addison has defined the following preferential subscriptions for movies: ($\{genre = comedy\}$, 0.9), ($\{genre = romance\}$, 0.9), ($\{genre = drama\}$, 0.8), ($\{genre = horror\}$, 0.6). She has also expressed her interest in receiving the top-2 events per period and that each period lasts 30 minutes. Assume further that the movie theaters which use the service publish the events e_1, e_2, ..., e_8 of Figure 6.6 in that order, at the time shown on top of each event. Figure 6.6 also shows the events that will be delivered to Addison for $\sigma = 0.5$. For the time period that begins at 20:00 and

ends at 20:30, the top-2 results are the events e_2 and e_4 because comedies and romances are ranked higher that drama movies and e_1 is older than e_4, while from 20:30 to 21.00 the top-2 results are the events e_7 and e_8 because e_5 and e_6 are older than e_7.

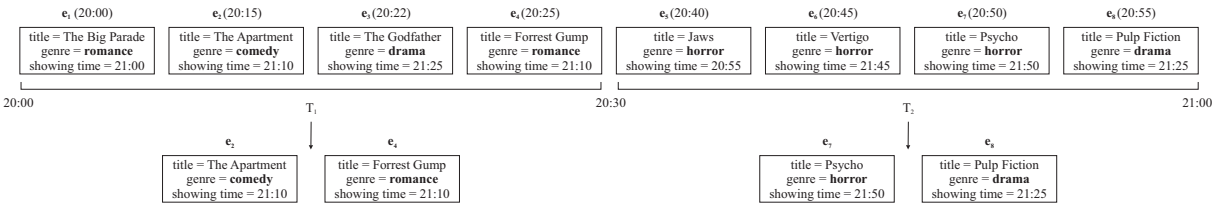


Figure 6.6: Periodic top-2 events for Addison ($T = 30$ min, $\sigma = 0.5$).

6.4.2 Sliding-Window Delivery

With periodic delivery, top-k computation starts anew at the beginning of each period. In contrast, with sliding window, top-k computation starts anew, each time a new event is published. In particular, we call *window* of length w an ordered list of w events, denoted $W = (e_1, e_2, \ldots, e_w)$ where e_i precedes e_{i+1} in the window, if and only if, no other event was published between them, that is, $\nexists\, e$, such that $tpub_{e_i} < tpub_e < tpub_{e_{i+1}}$. Let W_f be the window that includes the first w events published. If $W_i = (e_{i_1}, e_{i_2}, \ldots, e_{i_w})$, $i \geq f$, then $W_{i+1} = (e_{i_2}, \ldots, e_{i_w}, e)$, where e is the first event published after e_{i_w}. As a special case, before the first w events are published, the corresponding w-1 windows include only the events published so far and have length shorter than w. At the end of each window W_i, the k highest ranked matching events published within this window are forwarded to the users. Formally:

Definition 6.13 (Sliding-Window Top-k)**.** Let X be a user. Let $WS_e = \{W_m \mid e \in W_m\}$ be the set of w windows an event e belongs to. For each window W_m, let WS_m be the set of events in W_m. Let L_{W_m} be a subset of WS_m with k events that has the maximum $divrank(L_{W_m}, X)$ among all subsets of W_m with the same cardinality. If there are more than one such subsets, let L'_{W_m} be one with the maximum $\sum_{e_i \in L'_{W_m}} tpub_{e_i}$ among them. If again, there are more than one such sets, we randomly select one of them. Event e is forwarded, if and only if, $e \in L'_{W_m}$ for some $W_m \in WS_e$.

In our example, assume a window of length $w = 4$ and the published events of Figure 6.7. As shown in the figure, if Addison is again interested in the top-2 results with $\sigma = 0.5$, the first window W_1 returns its single event, i.e. e_1. The top-2 events of W_2 are e_1 and e_2 and since e_1 has already been sent to Addison, the only new result is e_2. W_3 contains no new results because dramas are less preferred than comedies and romances. The top-2 events of W_4 are e_2 and e_4, so e_4 is sent to Addison and so on.

In contrast to periodic delivery, the delivery rate is not constant, but depends on the relative order of the published events. When top-ranked events are computed based only on user preferences ($\sigma = 1.0$), we deliver at most one new event at each new window, as shown next.

Property 6.2. *When diversity is not used, between two consequent event-windows, at most one new event enters the top-k results.*

Proof. Assume a window W_q and its following window W_{q+1}, both of length w, and the two sets L_{W_q}, $L_{W_{q+1}}$ with the top-k events for W_q and W_{q+1} respectively. Since W_q and W_{q+1} have $(w-1)$ common events, let $W_q = (e_1, e_2, \ldots, e_w)$ and $W_{q+1} = (e_2, e_3, \ldots, e_{w+1})$. When e_{w+1} is published, e_1 leaves the window and one of the following holds:

- $e_1 \in L_{W_q}$, then $L_{W_{q+1}} = (L_{W_q} - \{e_1\}) \cup \{e'\}$, where e' is either e_{w+1} or e' was published in W_q and $e' \notin L_{W_q}$, or
- $e_1 \notin L_{W_q}$, then $L_{W_{q+1}} = L_{W_q}$ or $L_{W_{q+1}} = L_{W_q} - \{e'\}) \cup \{e_{w+1}\}$, where e' was published in W_q.

In any case, at most one event enters the set $L_{W_{q+1}}$. ∎

However, when diversifying events, the top-k events over W_{q+1} are not necessarily related with the top-k over W_q, since *divranks* are computed based not only on the (fixed) user preferences but also on the distances among the various candidate events. For example, a new highly preferable event may now disqualify more than one top-k events because it is very similar to them. This observation leads us to the following property:

Property 6.3. *When diversity is used, between two consequent event-windows, more than one new event can enter the top-k results.*

Proof. To illustrate this, let $e_1, \ldots, e_5$ be a series of events. e_1 is a comedy directed by W. Allen with rank 0.9, e_2 is a thriller directed by T. Burton with rank 0.9, e_3 is an A. Hitscock's thriller with rank 0.8, e_4 is a S. Spielberg's drama with rank 0.85 and finally, e_5 is a Q. Tarantino's drama with rank 0.9. Assume a window length $w = 3$, then $W_1 = (e_1)$, $W_2 = (e_1, e_2)$, $W_3 = (e_1, e_2, e_3)$, $W_4 = (e_2, e_3, e_4)$ and $W_5 = (e_3, e_4, e_5)$. Let $k = 2$ and $\sigma = 0.5$. W_1 will return e_1, W_2 will return e_2, W_3 will return no event, W_4 will return e_4 and W_5 will return both e_3 and e_5. ∎

An event may remain in the window and be delivered after as many as w other more recent events have been delivered. Thus, with sliding window, events may enter the top-k list in an order different from their publication order (see for example e_3 in Figure 6.7).

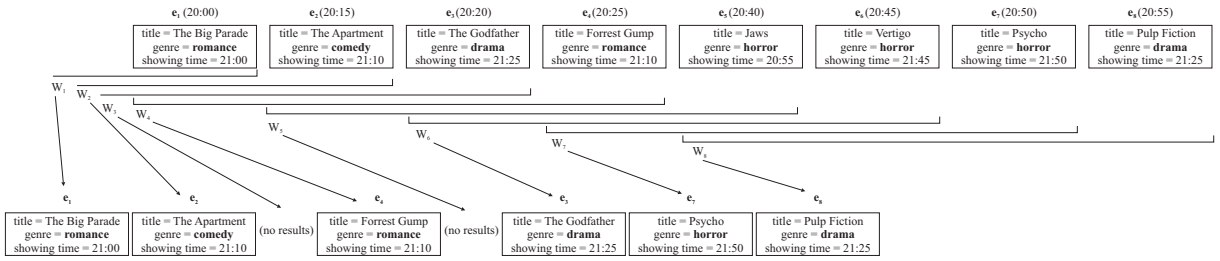


Figure 6.7: Sliding-window top-2 events for Addison ($w = 4$, $\sigma = 0.5$).

6.4.3 History-based Filtering

Delivery using history-based filtering considers each new event as it arrives and decides whether to deliver it or not, based on history, i.e. the last top-k events seen by the user. To refresh the events delivered, we assume that an event can remain in the top-k list for up to a window of w events.

Definition 6.14 (History-Based Top-k)**.** Let X be a user. Let e be an event published at time instant $tpub_e$, H be the set of the last top-k events delivered to the user and e' be the event published w events prior to e (the event that expires when e is published). Event e is delivered, if and only if, one of the following holds: (i) $e' \in H$, or (ii) $divrank((H - \{e_i\}) \cup \{e\}, X) \geq divrank(H, X)$ for some $e_i \in H$.

In our example, when Addison selects this policy, her top-2 events will be the ones shown in Figure 6.8 (we again assume a window of length $w = 4$ and $\sigma = 0.5$).

As with sliding-window, the total number of delivered events is not bounded by k. However, since only newly published events can be delivered to the users, at most one new event can enter the top-k results at each window, even with diversity.

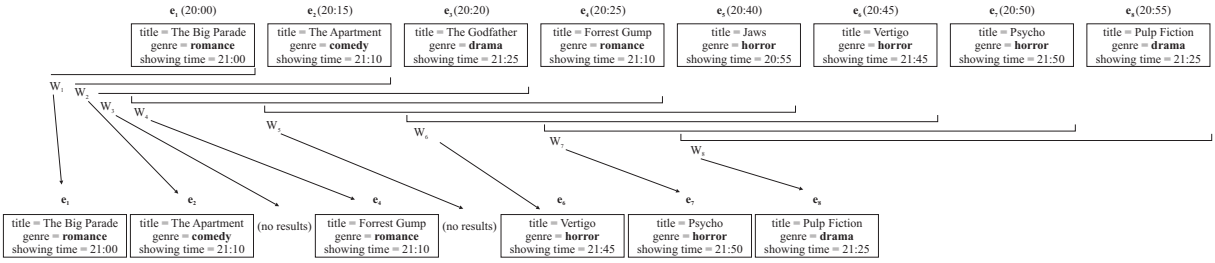


Figure 6.8: History-based top-2 events for Addison ($w = 4$, $\sigma = 0.5$).

6.5 The Event-Notification Service

In this section, we outline a method for matching events with subscriptions and computing event ranks. To this end, we introduce a *preferential subscription graph* for organizing our preferential subscriptions. We also show how to compute the top-k results for each delivery policy.

6.5.1 Event Matching

To reduce the complexity of the matching process between events and subscriptions, we organize the subscriptions using a graph similar to the *filters poset* data structure [26]. All subscriptions are organized in a directed acyclic graph, called *preferential subscription graph*, or PSG, whose nodes correspond to preferential subscriptions and edges to cover relations between them. Preferential subscriptions issued by different users which contain the same subscription are grouped together in a single graph node.

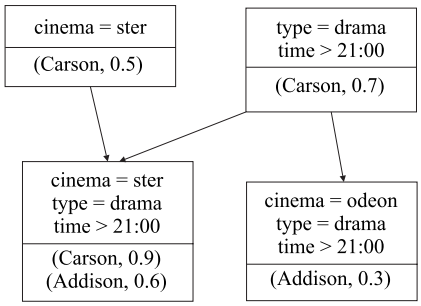


Figure 6.9: Preferential subscription graph example.

In particular, let P be the set of all preferential subscriptions, i.e. the preferential subscriptions defined by all users, and P_S be the set of all subscriptions in P. For each subscription $s_i \in P_S$, we maintain a set of pairs, called *PrefRank Set*, of the form (X, $prefrank_i^X$), where X is a user and $prefrank_i^X$ is the preference rank of X for s_i. A subscription s_i is associated with the pair (X, $prefrank_i^X$), if and only if, a preferential subscription $ps_i^X = (s_i, prefrank_i^X)$ exists in P. For each $s_i \in P_S$, we define the *PrefRank Set* as the set $PR_i = \{(X, prefrank_i^X) \mid (s_i, prefrank_i^X) \in P\}$. Formally:

Definition 6.15. (Preferential Subscription Graph). Let P be a set of preferential subscriptions and P_S the set of all subscriptions in P. A Preferential Subscription Graph $PSG_P(V_P, E_P)$ is a directed acyclic graph, where for each different $s_i \in P_S$, there exists a node v_i, $v_i \in V_P$, of the form (s_i, PR_i), where PR_i is the PrefRank Set of s_i. Given two nodes v_i, v_j, there exists an edge from v_i to v_j, $(v_i, v_j) \in E_P$, if and only if, s_i covers s_j and there is no node v'_j such that s_i covers s'_j and s'_j covers s_j.

For example, Figure 6.9 depicts the PSG of the preferential subscriptions of two users, Carson and Addison. Carson has specified subscription $s_1 = \{cinema = ster, genre = drama, time > 21{:}00\}$ with preference rank 0.9, $s_2 = \{genre = drama, time > 21{:}00\}$ with 0.7 and $s_3 = \{cinema = ster\}$ with 0.5. Addison's subscriptions are s_1 with preference rank 0.6 and $s_4 = \{cinema = odeon, genre = drama, time > 21{:}00\}$ with 0.3.

When a new event e arrives to the event-notification service, we traverse the PSG to locate all matching subscriptions in a breadth-first manner starting from the root nodes. In some cases, it is not necessary to walk through all nodes of the graph. We may safely ignore a node v with subscription s for which there is no other node v' with subscription s', such that s' covers s and $s' \succ e$. This means that whenever e does not match a specific node of the PSG, its whole sub-tree can be ignored. This way, entire paths of the graph can be pruned. For example, in Figure 6.9, if an incoming event is not covered by $\{cinema = ster\}$, then it is certainly not covered by $\{cinema = ster, genre = drama\ time > 21:00\}$ and this subscription does not have to be checked against the event.

However, if the incoming event is covered by a node v of the PSG, we have to check the event against other nodes in v's sub-tree, to retrieve their preference rank, since it is possible that some of these nodes may be more specific to the event than v. In our example, for an event $e = \{cinema = ster, genre = drama, time = 21{:}30\}$, s_3 is more

specific than s_1 for Carson. Therefore, even if $s_1 \succ e$, we have to continue traversing the PSG (note that in a traditional publish/subscribe system that would not be necessary). To avoid unnecessary traversals, we associate each entry of v's *PrefRank Set* with a status bit. This bit is set to 1, if the subscriber of the entry can also be found in some other node v' covered by v in the PSG and to 0 otherwise.

For each subscriber X associated with at least one subscription covering the event e, we compute the rank of the event. In our current work, we assume that preference ranks associated with subscriptions are indicators of positive interest, thus, we use as the aggregation function $\mathcal{F}$ the maximum value of the preference ranks of the covering subscriptions. Given that an event e is covered by m subscriptions $s_1, s_2, \ldots, s_m$ of user X, $rank(e, X) = \max\{prefrank_1^X, prefrank_2^X, \ldots, prefrank_m^X\}$. After this matching process, some information about the event (like the computed rank or the content) are stored according to the delivery policy used, as described next.

6.5.2 Event Delivery

Typically, publish/subscribe systems are stateless, in that, they do not maintain any information about previously delivered events. However, to provide users with the current top-ranked matching events, depending on the delivery policy, we may need to maintain some information about previously delivered events as well as buffer some published events prior to their final delivery or dismissal.

Periodic and Sliding-Window Delivery In the case of periodic delivery, the server needs to buffer all events that match at least one subscription in its PSG until the end of the period in which they were published and their corresponding ranks. At the end of the period, the top-k results for all users are computed using Algorithm 17, having as input all the events matched during the period. Analogously, in sliding-window delivery, we buffer the content and ranks of the w most recently published events, since an event may be forwarded to a user at some point after its publication time. In this case, the input set to Algorithm 17 is the set of events in the current window.

History-Based Filtering With history-based filtering, we need to maintain some information about previously sent top-ranked events. If we do not consider diversity, we do not need to maintain the actual content of the matched events, it suffices to buffer just the preference rank of the current top-k events. We also need to store an expiration counter along with each buffered event rank to determine when the event expires and disregard it. The counter is initialized to w and is decreased by one every time a new event is published. An element is removed from the buffer, when its counter becomes 0. A newly published matching event e is delivered, if and only if, (i) an event in the current top-k expires when e arrives or (ii) e is better than the worst event in X's current top-k. In this case, the worst event is disregarded and replaced by e.

History-Based Filtering Event Delivery

Input: A new event e, a buffer b of previously published matching events, a preferential subscription graph PSG, a subscriber X and the number k of desired top-results (and possibly a diversification factor σ).

Output: Whether e should be forwarded to X or not.

```
1: begin
2: result ← false;
3: b ← b ∪ {e};
4: TOP_init ← the current top-k results;
5: TOP ← TOP_init− the event that has just left the window (if any);
6: if |TOP| < k then
7:    result = true;
8: else
9:    if not diversify then
10:      find the lowest rank r_low in TOP;
11:      if rank(e, X) > r_low then
12:         result = true;
13:   else
14:      DIVR_init = divrank(TOP_init, X);
15:      for all e_i ∈ TOP do
16:         TOP_i ← {TOP − e_i} ∪ {e};
17:         DIVR_i = divrank(TOP_i, X);
18:      DIVR_final = max{DIVR_1, ..., DIVR_|TOP|};
19:      if DIVR_final > DIVR_init then
20:         result = true;
21:         TOP ← {TOP − e_final} ∪ {e};
22: return result;
23: end
```

Algorithm 18: History-Based Filtering Event Delivery

If we consider diversity, we also need to store the content of the events in the buffer for computing their distances with any new event. As before, if an event in the current top-k expires when a new event e arrives, e is forwarded to X. Otherwise, we swap each event e_i in the buffer with the new event e to produce a number of k candidate sets. Then, we compute the new diversity-aware set rank (*divrank*) for each candidate set. If some of these new candidate sets have larger *divrank* than the current top-k results, then we forward e to the user and insert e in the buffer in the place of the event e_i that corresponds to the candidate set with the maximum *divrank*. The process described above is summarized in Algorithm 18.

6.6 Evaluation

To evaluate our approach, we have extended the SIENA event notification service [6], a multi-threaded publish/subscribe system, to include preferential subscriptions with diversity and ranked event delivery. We refer to our prototype as PrefSIENA. PrefSIENA is available for download [5].

6.6.1 System Description

To evaluate the performance of our model, we use a real movie-dataset [3], which consists of data derived from the Internet Movie Database (IMDB) [1]. The dataset contains information about 58788 movies. For each movie, the following information is available: title, year, budget, length, user rating (rating), MPAA rating (mpaa) and genre(s).

Publishers generate publications by randomly selecting m_P movies and creating a new event for each of them consisting of the title, year, length, rating, MPAA and genre(s) of the movie. Publications are produced at a constant rate. Each subscriber generates m_S subscriptions, each of which is generated independently from the others. We select a number of the available attributes to appear in a subscription based on a zipf distribution, i.e. some attributes are more popular than others. The value of each attribute is also generated using a zipf distribution, so that some values are more common. Preferences are generated by associating preference ranks in $[0, 1]$ with the generated subscriptions. Those ranks have an average value around 0.5. In general, most specific publications get higher ranks.

Event delivery is performed following either one of our three delivering policies, i.e. the periodic, the sliding-window and the history-based filtering ones.

6.6.2 Experiments

We perform a number of different experiments. First, we evaluate the performance of our diversity heuristic. Then, we evaluate the number and quality of the events delivered to the users using PrefSIENA and SIENA. We also evaluate the overheads introduced by ranking and diversifying.

Heuristic Performance

To evaluate the performance of our diversity heuristic, we compare it against the brute-force method that finds optimal solutions. We compare these methods both in terms of the quality of produced results as well as the required time to produce them. The complexity of both methods depends on the number n of candidate events to choose from and on the required number k of events. We experiment with a number of different values for n and k. However, the high complexity of the brute-force algorithm prevents us from using large values for these two parameters. Therefore, we limit our study to $n = 10, 20, 30$ and

Table 6.1: Diversity Heuristic vs Brute-force performance.

n	k	Heuristic		Brute-force	
		Diversity	Time	Diversity	Time
10	4	0.873	16	0.917	41
	8	0.846	26	0.851	42
20	4	0.905	29	0.917	384
	8	0.850	32	0.866	43065
	12	0.820	37	0.832	99608
	16	0.811	58	0.814	6784
30	4	0.929	31	1.000	1987
	8	0.881	38	0.923	1967339
	12	0.870	46	0.889	51652649
	16	0.859	57	0.874	162827625
	20	0.846	69	0.857	50641750

$k = 4, 8, 12, 16, 20$. The results of our experiment are summarized in Table 6.1 (time is measured in milliseconds).

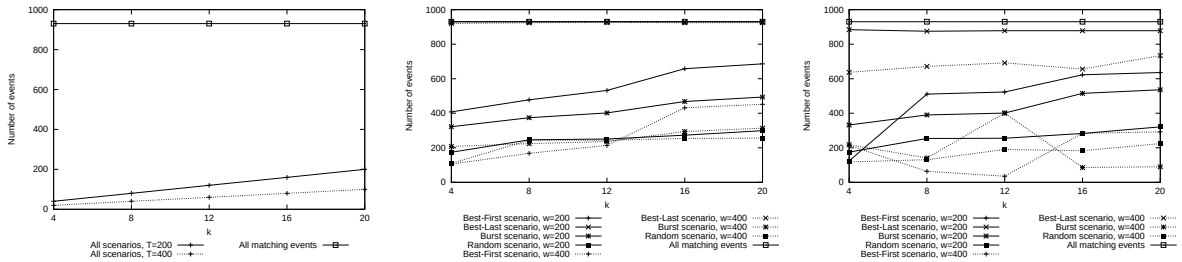
The complexity of the brute-force method is so high that even with the relatively small values of $n = 30$ and $k = 8$, the required time climbs up to 1967339 ms (≈ 0.5 hour). Our heuristic required only 38 ms in this case. This reduction in time complexity comes at the cost of decreased diversity of the results. However, this reduction is only marginal, as the set diversity of the results produced by the heuristic is decreased by less than 1% in all cases.

Number and quality of delivered events

One of the reasons that motivated ranked delivery was the need to reduce the large amount of events delivered to users in a traditional publish/subscribe system. Therefore, in this set of experiments, we first measure the total number of delivered events and then evaluate their quality in terms of average rank and diversity.

Since the number and quality of events depend on the order of publications with regard to their ranks, we consider a number of different event-scenarios. In particular, in the "Best-First" scenario, the highest-ranked events are published first, while in the "Best-Last" scenario, these events are published after the lower-ranked ones. In the "Burst" scenario, we consider that there exist bursts of highly-ranked events at specific moments in time and finally, in the "Random" scenario, high and low ranked events are interleaved. For comparison, besides top-k delivery, we also consider the case in which all matching events are delivered to the users, as in the case of a traditional publish/subscribe system. All scenarios consist of 2000 events out of which 930 match the subscriptions.

Number of delivered events. We measure the number of events delivered to a specific subscriber using PrefSIENA as a function of the number k of the top results the



(a) Periodic delivery policy (b) Sliding-window delivery policy (c) History-based delivery policy

Figure 6.10: Total number of delivered events ($\sigma = 1.0$ - no diversity).

subscriber is interested in. We first consider the case where events are selected based solely on event ranks (i.e. $\sigma = 1.0$). In Figure 6.10a, we show the number of delivered events for the periodic delivery policy. For comparison reasons, we consider a constant rate of publications and run this experiment for periods with $T = 200$ and $T = 400$ events. We see that the number of delivered events does not depend on the used scenario, since at each period this number is bounded by k. Therefore, we achieve a constant rate of event delivery. On average, the number of events delivered by PrefSIENA ranges from 2.2% to 21.5% of all matching events for the various values of k and T.

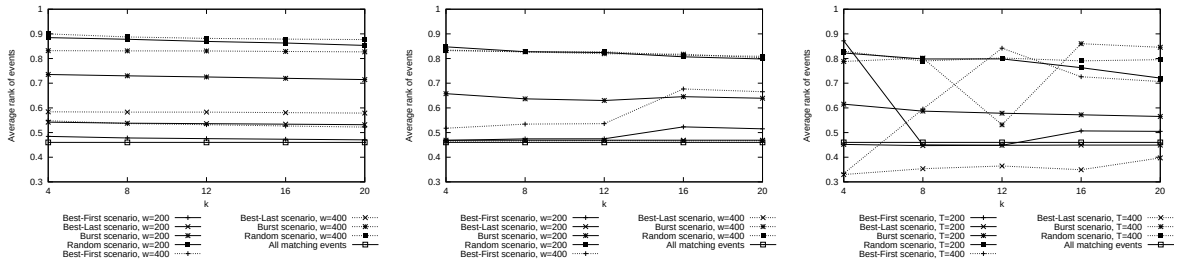
In Figures 6.10b and 6.10c, we present the total number of delivered events for the sliding-window and history-based filtering policies. We run the experiment for all of the above scenarios and use window lengths of $w = 200$ and $w = 400$ events. We see that when those policies are used, the number of delivered events depends not only on k and w but also on the input. For the sliding-window policy, we observe that a larger window size leads to the delivery of fewer events. The best pruning is achieved when the "Random" scenario is used. This happens because in this case, there are always some highly preferable events in the current window to filter out less preferable ones. The worse pruning, as expected, is observed in the case of the "Best-Last" scenario, since in this case every new event is better than the old ones and is thus forwarded to the user. In the case of the history-based filtering, the "Best-Last" scenario also has the worse pruning while the "Random" one achieves the greatest pruning. Once again, a larger window length results in greater reduction of delivered events.

When diversity is also a factor for choosing which events to deliver, only the sliding-window and history-based filtering policies are affected. In the periodic policy, while the delivered events may actually be different than in the no-diversity case, their total number remains the same. In the sliding-window case, the total number of delivered results is slightly larger due to Property 6.3, while in the history-based case the number depends on the input (we omit the relative figures).

Quality of delivered events. We also run a set of experiments to evaluate the quality of the delivered events. We characterize quality based on two factors: (i) the average rank of delivered events and (ii) their diversity.

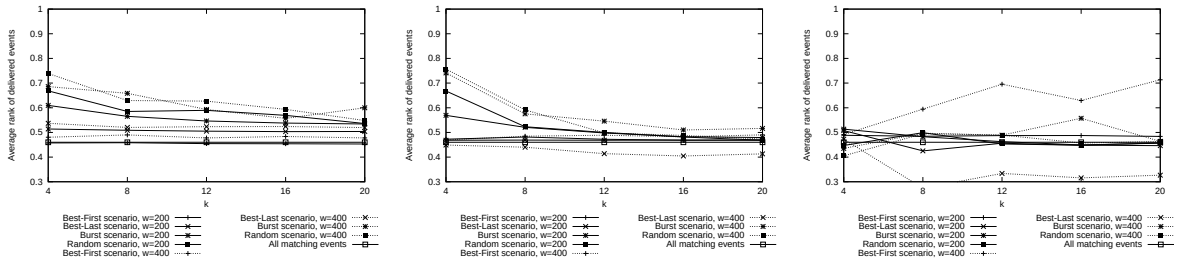
Figure 6.11 depicts the average rank of all the delivered events for the various timing

policies and scenarios when $\sigma = 1.0$ (no diversity case). Average rank is computed over all events delivered in each delivery policy (as opposed to over the same number of top-ranked events). Generally, we observe that the average rank depends on the input. The average rank of all matching events is 0.46. In PrefSIENA, even though in the presence of many high-ranked events some of them may fail to appear in the top-k results, the average rank is larger than that in all cases. When diversifying the events, there is a decrease of the average rank, since diverse events may have lower ranks, as expected (Figures 6.12 and 6.13). The average rank of events decreases along with k, since for large values of k more events are delivered to the users. This decrease is more evident when diversity is used.



(a) Periodic delivery policy (b) Sliding-window delivery policy (c) History-based delivery policy

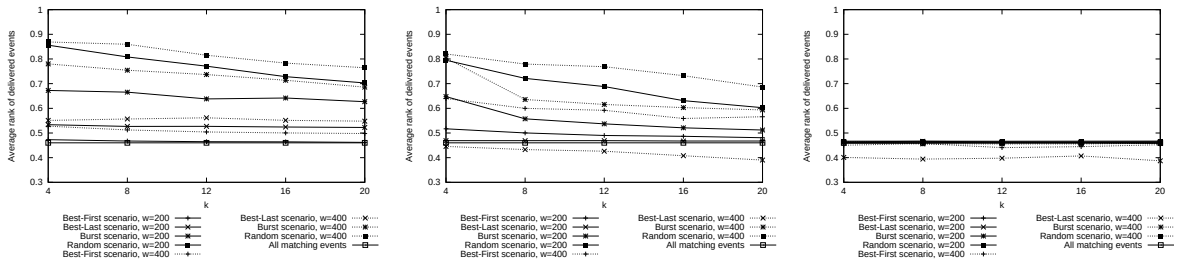
Figure 6.11: Average rank of delivered events ($\sigma = 1.0$ - no diversity).



(a) Periodic delivery policy (b) Sliding-window delivery policy (c) History-based delivery policy

Figure 6.12: Average rank of delivered events ($\sigma = 0.0$ - no ranking).

In Figure 6.14, we measure the average diversity of the events that are forwarded to a user for the "Random" scenario. Average diversity is computed over the events delivered in each period or window. We run this experiment for different period and window lengths using our diversification methods with $\sigma = 1.0$ (no diversity case), $\sigma = 0.5$ and $\sigma = 0.0$ (no ranking case). We see that the produced results do indeed exhibit a higher diversity when they are chosen based not only on their ranks but also on their distance from each other. This increase is larger for smaller values of k. Similar behavior can be observed for the other scenarios as well. Due to space limitations, we omit the related figures.



(a) Periodic delivery policy (b) Sliding-window delivery policy (c) History-based delivery policy

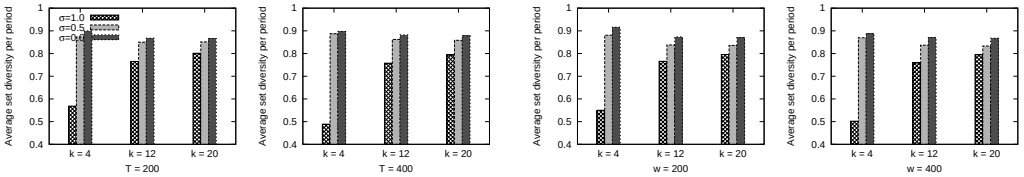
Figure 6.13: Average rank of delivered events ($\sigma = 0.5$).

Performance

Finally, we perform a number of experiments to evaluate the performance of PrefSIENA. There are two sources of extra overhead for implementing ranked delivery of events. First, to compute the importance of a new event, we have to locate all matching subscriptions, while in traditional publish/subscribe systems it suffices to locate the most general one. Second, there is also the overhead of maintaining state for previously forwarded events and performing computations to decide whether a new event belongs in the diverse top-ranked results or not.

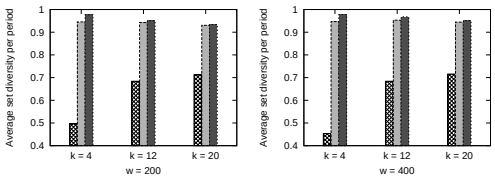
The matching overhead depends on the relations among the various user subscriptions. More specifically, the overhead is more evident when users issue many subscriptions that cover each other, i.e. users refine their previously made subscriptions. To compute this overhead, we perform the following experiment: we construct a number of profiles in which a percentage c of user subscriptions are covered by some other subscription. We also construct a number of scenarios in which a percentage m of the published events match the user subscriptions. In Table 6.2, we see the number of PSG nodes checked during the execution of each scenario for each of the user profiles in SIENA and PrefSIENA, with $c = 0\%, 10\%, 30\%, 50\%$ and $m = 0\%, 10\%, 20\%$.

To measure the time overhead introduced by the ranking and diversifying algorithms, we measure the time between the publication and the delivery of each event (Figure 6.15). In the periodic policy, the sequence of the published events influences the freshness of the delivered ones. For example, if high-ranked events are published towards the end of a period, they will reach the user earlier than if they are published at the beginning. As expected, a larger period length results in larger delays between publication and delivery. In the sliding-window delivery policy, a larger window length increases the average delivery time. This happens (i) because an event remains in the window for longer and therefore, it has more opportunities to enter the top-k results and (ii) because the complexities of the ranking and diversifying algorithms depend on the window size. In the history-based filtering delivery policy, the freshness of data does not depend much on the scenario, but is rather more influenced by the size of the window.



(a) Periodic delivery policy

(b) Sliding window delivery policy



(c) History-based delivery policy

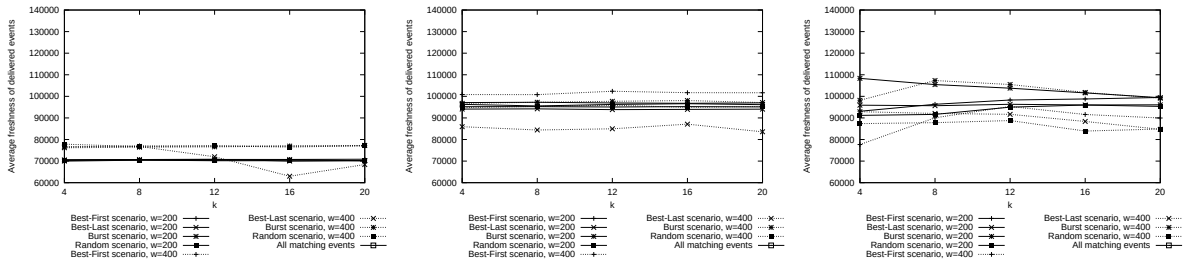
Figure 6.14: Average diversity - random scenario.

6.7 Related Work on Ranking in Publish/Subscribe Systems and Diversity

Although there has been a lot of work on developing a variety of publish/subscribe systems, there has been only little work on the integration of ranking issues into publish/subscribe. Recently, in [91], the problem of ranked publish/subscribe systems is also considered. However, the problem is viewed in a different way. In a sense, the authors consider the "reverse" or "dual" problem. Instead of locating the most relevant events to each subscription, the authors aim at recovering the most relevant matching subscriptions to a published event. Subscriptions are modeled as sets of interval ranges in some dimensions and events as points that match all the intervals that they stab. Another work that also deals with the problem of ranked publish/subscribe is [101]. In the proposed model, a subscriber receives the k most relevant events per subscription within a window w which can be either time-based or event-based. For each user subscription, a queue is maintained. This queue buffers those events that are relevant to the subscription and have a high probability to enter the top-k result at some point in the future. The focus is on efficiently maintaining this buffer queue. Here, we aim at specifying and computing event ranks. [137] considers the case where only a subset of top-ranked publishers provide notifications for a specific query. These publishers are ranked according to the similarity

Table 6.2: PrefSIENA matching overhead.

m	c	SIENA	PrefSIENA
0%	0%	10000	10000
	10%	9000	9000
	30%	7000	7000
	50%	5000	5000
10%	0%	10000	10000
	10%	9000	9010
	30%	7000	7010
	50%	5000	5010
20%	0%	10000	10000
	10%	9000	9020
	30%	7000	7020
	50%	5000	5010



(a) Periodic delivery policy (b) History-based delivery policy (c) Sliding window delivery policy

Figure 6.15: Average freshness of delivered events ($\sigma = 0.5$).

of their past publications to the query. Similarity is computed via IR techniques. [123] suggests using an extensive preference model to enhance expressiveness of subscription matching, while [89] proposes an approximate matching mechanism so that relevant events are delivered even if they do not match exactly the users' subscriptions.

In terms of diversity, in [136], a method for topic diversification is proposed for recommendations. The intra-list similarity metric is introduced to assess the topical diversity of a given recommendation list. An algorithm that considers the candidate items' scores is provided for creating lists with small intra-list similarity. The notion of diversity is also explored in [122]. Motivated by the fact that some database relation attributes are more important to the user, a method is proposed where a recommendation list consisting of database tuples is diversified by first varying the values of higher priority attributes before varying the values of lower priority ones. In case the tuples are associated with scores, a scored variation of diversity always picks tuples with higher scores first.

6.8 Summary

Our overall goal, in this chapter, has been to increase the quality of events received by the users of publish/subscribe systems in terms of their importance or relevance and diversity. Ranking events by importance is achieved by letting users express preferences along with their subscriptions. Events that match more preferable subscriptions are ranked higher than events that match less preferable ones. For ranking an event, we also take into account how different the event is from the other top-ranked ones so that the overall diversity among the event notifications is increased. We have examined a number of policies with regards to the time range over which the top-k events are computed, namely a periodic, a sliding-window and a history-based one.

The results presented in this chapter also appear in [40, 41].

Chapter 7

Conclusions

The goal of this thesis was the study of preferences in data management systems. To this end, we proposed a context-dependent model for preferences and appropriate data structures and algorithms for managing contextual preferences. We also explored the integration of preferences to achieve personalized keyword search in relational database systems and personalized delivery in publish/subscribe systems. We conclude, in this chapter, with a summary of our primary contributions (Section 7.1) and directions for future research (Section 7.2).

7.1 Summary of Contributions

The technical contribution of this thesis was centered around two main axes. The first axis focused on modeling issues. We introduced a model for contextual preferences and focused on managing such preferences. We also addressed the problem of scoring database tuples using our contextual preferences and provided a solution which was based on selecting and pre-computing representative rankings. The second axis was centered on integrating preferences in data management systems. We proposed personalized keyword search in relational database systems and personalized delivery in publish/subscribe systems through user-defined preferences.

As a side contribution of this thesis, we also consider an extensive survey of the use of preferences in databases. We considered ways in which the general notion of preference may be interpreted in a database system and we classified approaches based on certain criteria. We identified the following main axes and organized our study around them: (*i*) preference representation, (*ii*) preference composition and (*iii*) preferential query processing. We also discussed preference learning approaches.

We proposed annotating database preferences with contextual information. Context was modeled using a set of multidimensional context parameters that take values from hierarchical domains, thus, allowing different levels of abstraction for the captured context data. We also formulated the problem of context resolution, as the problem of selecting appropriate preferences for personalizing database queries based on context. To realize context resolution, we proposed two data structures, namely the *preference graph* and the *profile tree*, that allow for a compact representation of the context-dependent preferences.

To evaluate the usefulness of our model, we performed usability studies. Our studies showed that annotating preferences with context improves the quality of the retrieved results considerably. The burden of having to specify contextual preferences is reasonable and can be reduced by providing users with default preferences that they can edit. We also performed a set of experiments to evaluate the performance of context resolution using both real and synthetic datasets. The proposed data structures were shown to improve both the storage and the processing overheads. In general, the profile tree is more space-efficient than the preference graph. It also clearly outperforms the preference graph in the case of exact matches. The main advantage of the preference graph is the possibility for an incremental refinement of a context state. In particular, at each step of the resolution algorithm, we get a state that is closer to the query one. This is not possible with the profile tree.

To improve performance, we introduced a suite of techniques for quickly providing users with data of interest based on contextual preferences. We proposed performing pre-processing steps to construct representative rankings of database tuples. To form such rankings, we focused on creating groups of similar preferences and producing a ranking for each group, first by considering as similar the preferences that have similar contexts. To group preferences with similar contexts, we considered a contextual clustering method that exploits the hierarchical nature of context parameters. Our method can be applied to both quantitative and qualitative preferences. We also presented a complementary method for grouping preferences according to the similarity of the scores that they produce. This method takes advantage of the quantitative nature of preferences to group together contextual preferences that have similar predicates and scores. The method is based on a novel representation of preferences through a predicate bitmap table whose size depends on the desired precision for the resulting scoring. We evaluated our approach using both real and synthetic data sets and presented experimental results showing the quality of the scores attained using our methods.

We also proposed personalizing keyword search through user preferences and provided a formal model for integrating preferential ranking with database keyword search. By extending query-relevance ranking with preferential ranking, users are expected to receive results that are more interesting to them. To further increase the quality of results, we suggested selecting k representative results that cover many user preferences and exhibit small overlap. We presented algorithms that extend current schema-based approaches for keyword search in relational databases to incorporate preference-based ranking and

top-k representative selection. We evaluated both the efficiency and effectiveness of our approach. Our performance results showed that our sharing-results algorithm significantly improves the execution time over the baseline algorithm. Furthermore, the overall overhead for preference expansion and diversification is reasonable. Our usability results indicated that users receive results more interesting to them when preferences are used.

Finally, we addressed the problem of increasing the quality of events received by the users of publish/subscribe systems in terms of their importance and diversity. Ranking events by importance is achieved by letting users express preferences along with their subscriptions. Events that match more preferable subscriptions are ranked higher than events that match less preferable ones. For ranking an event, we also took into account how different the event is from the other top-ranked ones, so that, the overall diversity among the event notifications is increased. We examined a number of policies with regards to the time range over which the top-k events are computed, namely a periodic, a sliding-window and a history-based one. We implemented a prototype, termed PrefSIENA [5]. PrefSIENA extends SIENA [6], a popular publish/subscribe middleware system, with preferential subscriptions, delivering policies and diversity towards achieving top-k event delivery. We presented a number of experimental results to assess the number of events delivered by PrefSIENA with respect to the original SIENA system as well as their rank and diversity. We also reported on the overheads of supporting diversity-aware top-k delivery.

7.2 Future Research Directions

In this section, we provide directions for future research on issues that are still open and are the subject of our ongoing and future work. We distinguish between short term plans that consist mainly of extensions to our work and long term plans that highlight open research challenges.

7.2.1 Short Term Plans

Context Relaxation: When dealing with the *context resolution* problem, we have focused on relaxing the context of a query, so that, there are enough preferences whose associated context match that of the query. In general, we have so far mainly considered relaxing a hierarchical context value by using a more general one. However, apart from *upwards relaxation*, a context value may be relaxed *downwards* by replacing the value by a set of more specific ones or *sideways* by replacing the value by sibling values in the hierarchy. Given all these possible relaxation types, appropriate distance metrics that exploit the number of relaxed context values of a context state and the associated depth of such relaxations need to be defined. In this direction, and as a first step, in [114] we employ measures to study how well a context state matches a relaxed one.

Novelty in Publish/Subscribe Delivery: Recent research in publish/subscribe systems has suggested that event matching should be best effort by associating some form of ranking to the matching process. In our work, reported in this thesis, we have focused on ranked-based publish/subscribe delivery based on preferences, where events are ranked based on user interests, and diversity, where events are ranked so that users receive events with different content. Recently, in [105], we present a first approach to another dimension of ranking, that of novelty. Our interpretation of novelty is that an event is novel if it matches a subscription that has been rarely matched in the past. This form of novelty is desirable for various reasons, such as making rare events visible and allowing expressing an information need with various levels of detail.

Hierarchies in Contextual Keyword Preferences: An interesting direction in preferential keyword search is extending keyword queries and preference specifications to include keywords that take values from hierarchical domains along the lines of our previous work [113]. For instance, when the search for directors from Greece returns a small number of results, one could extend the search to European directors. Similarly, when there is no related preference for Greek directors, one could use preferences for directors from other European countries.

Recommendations in Relational Databases: In general, recommendation methods are categorized into: (*i*) *content-based*, that recommend items similar to those the user has preferred in the past, (*ii*) *collaborative*, that recommend items that similar users have liked in the past and (*iii*) *hybrid*, that combine content-based and collaborative ones. Recently, we have proposed extending relational database systems with a recommendation functionality. In particular, motivated by the way recommenders work, we have considered, along with the results of each query, "recommending" to users additional results of potential interest. We call such results "*You May Also Like*" or YMAL results. In [107], we have provided a taxonomy of three fundamentally different approaches to computing YMAL results. The first one, termed *current-state*, uses the results and the schema of the current query and the database. The second one, termed *history-based*, is similar to traditional recommendation systems. It uses the past history of user queries to suggest tuples that are results of either similar past queries or results of queries posed by similar users. The last one, termed *external sources*, considers using information from resources external to the database, such as the web. In our current work, we mainly focus on the current-state approach, since we believe that using the results of a query to recommend other possibly interesting results is a challenging direction for recommendations.

7.2.2 Long Term Plans

Hybrid Preference Models: The preference models proposed so far by the research community follow either the qualitative or the quantitative approach. Using qualitative

preferences, we cannot distinguish how much better a query answer is compared to another. We could exploit ideas from both philosophies. For instance, we could express preferences, such as thriller movies are preferred over dramas with score 0.7 and dramas are preferred over comedies with score 0.5. Defining such hybrid preference models is challenging. For example, given the preferences above, what is the relationship between thriller and comedy movies? Furthermore, people naturally express their preferences in either way (e.g., "I like comedies a lot" and "I like comedies more than dramas"). We could also define a hybrid preference model that allows expressing both qualitative and quantitative preferences. Then, several interesting issues arise: How can we combine hybrid preferences? How can we rank query results, using both a qualitative or a quantitative approach?

Group or Social Preferences: The need for group or social preferences comes along with a number of scenarios. For example, recommend the most preferred restaurants or movies for friends taking into account the preferences of each individual. A natural question is: How can individual preferences be aggregated into social preferences? How is a social preference formally defined? In general, there are two fundamental ways to create group rankings: (*i*) compute for each user a ranking of results based only on the user's preferences and merge these individual rankings to compute an overall ranking, i.e., a ranking for the group, and (*ii*) aggregate the preferences of all users into a single profile that is subsequently used to provide an overall ranking. To deal with inconsistencies or conflicts of interest, we need to define different aggregation policies, such as acceptance, tolerance and disagreement. Ranking aggregation has been recently discussed in the context of group recommendations [14].

Preferential Social Search: Usually, in social web systems, users share resources, such as photos, research papers, answers to questions, blogs and other personal information. Since such systems become extremely popular and thus, an important component of the web, exploring new ways of searching the social graph is challenging. Building upon our previous work, we envision a social system, where users are allowed to restrict their view of the social graph based on the current context and related preferences. More specifically, users will be able to query the social graph and be presented with a diversified subgraph relevant to their interests. As the time passes, results will be adapted to the user's new context. For instance, a query about friends' news submitted on Monday morning may begin returning information about various activities of the user's colleagues. By the evening of the same day, the same query may be presenting information about the user's friends and family. From a different perspective, social systems can be used for deducing user preferences. By exploiting the amount of personal information currently available over the social web, interesting knowledge about users can be elicited.

Bibliography

[1] *Internet Movies Database*. Available at http://www.imdb.com/.

[2] *MovieLens 2003*. Available at http://www.grouplens.org/data.

[3] *Movies dataset*. Available at http://had.co.nz/data/movies.

[4] *PerK: Experimental Info (Source code and datasets)*. Available at http://www.cs.uoi.gr/∼kstef/PerK.

[5] *PrefSIENA*. Available at http://www.cs.uoi.gr/∼mdrosou/PrefSIENA.

[6] *SIENA*. Available at http://serl.cs.colorado.edu/∼serl/dot/siena.html.

[7] *Stanford Movies Dataset*. Available at http://infolab.stanford.edu/pub/movies/.

[8] *TCP-H Dataset*. Available at http://www.tcp.org.

[9] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Trans. Knowl. Data Eng.*, 17(6):734–749, 2005.

[10] R. Agrawal, R. Rantzau, and E. Terzi. Context-sensitive ranking. In *SIGMOD*, pages 383–394, 2006.

[11] R. Agrawal and E. L. Wimmers. A framework for expressing and combining preferences. In *SIGMOD*, pages 297–306, 2000.

[12] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

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

[14] S. Amer-Yahia, S. B. Roy, A. Chawla, G. Das, and C. Yu. Group recommendation: Semantics and efficiency. *PVLDB*, 2(1):754–765, 2009.

[15] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, pages 564–575, 2004.

[16] M. Bazire and P. Brézillon. Understanding context before using it. In *CONTEXT*, pages 29–40, 2005.

[17] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.

[18] C. Bolchini, C. Curino, E. Quintarelli, F. A. Schreiber, and L. Tanca. A data-oriented survey of context models. *SIGMOD Rec.*, 36(4):19–26, 2007.

[19] J.-C. Borda. *Mémoire sur les élections au scrutin*. Histoire de l'Académie Royale des Sciences, 1781.

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

[21] C. Boutilier, R. I. Brafman, H. H. Hoos, and D. Poole. Reasoning with conditional ceteris paribus preference statements. In *Proc. of the Sym. on Uncertainty in AI*, pages 71–80, 1999.

[22] J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *UAI*, pages 43–52, 1998.

[23] P. Brown, J. Bovey, and X. Chen. Context-aware applications: From the laboratory to the marketplace. *IEEE Personal Communications*, 4(5):58–64, 1997.

[24] C. Buckley and E. M. Voorhees. Retrieval evaluation with incomplete information. In *SIGIR*, pages 25–32, 2004.

[25] C. J. C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. N. Hullender. Learning to rank using gradient descent. In *ICML*, pages 89–96, 2005.

[26] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Syst.*, 19:332–383, 2001.

[27] C. Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. Finding k-dominant skylines in high dimensional space. In *SIGMOD*, pages 503–514, 2006.

[28] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic information retrieval approach for ranking of database query results. *ACM Trans. Database Syst.*, 31(3):1134–1168, 2006.

[29] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *ICDT*, pages 56–70, 1997.

[30] G. Chen and D. Kotz. A Survey of Context-Aware Mobile Computing Research. Technical Report TR2000-381, Dartmouth College, Computer Science, November 2000.

[31] J. Chomicki. Querying with intrinsic preferences. In *EDBT*, pages 34–51, 2002.

[32] J. Chomicki. Preference formulas in relational queries. *ACM Trans. Database Syst.*, 28(4):427–466, 2003.

[33] J. Chomicki. Semantic optimization techniques for preference queries. *Inf. Syst.*, 32(5):670–684, 2007.

[34] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pages 717–719, 2003.

[35] W. W. Cohen, R. E. Schapire, and Y. Singer. Learning to order things. *J. Artif. Intell. Res. (JAIR)*, 10:243–270, 1999.

[36] J. A. N. Condorcet. *Éssai Sur L' application De L' analyse á La Probabilité Des Décisions Rendues á La Pluralité Des Voix*. Kessinger Publishing, 1785.

[37] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *VLDB*, pages 451–462, 2006.

[38] J. P. Delgrande, T. Schaub, and H. Tompits. A framework for compiling preferences in logic programs. *TPLP*, 3(2):129–187, 2003.

[39] A. K. Dey. Understanding and using context. *Personal Ubiquitous Comput.*, 5(1):4–7, 2001.

[40] M. Drosou, E. Pitoura, and K. Stefanidis. Preferential publish/subscribe. In *PersDB*, pages 9–16, 2008.

[41] M. Drosou, K. Stefanidis, and E. Pitoura. Preference-aware publish/subscribe delivery with diversity. In *DEBS*, pages 1–12, 2009.

[42] E. Erkut. The discrete p-dispersion problem. *European Journal of Operational Research*, 46(1):48 – 60, 1990.

[43] E. Erkut, Y. Ülküsal, and O. Yeniçerioglu. A comparison of p-dispersion heuristics. *Computers & OR*, 21(10), 1994.

[44] M. Ester, J. Kohlhammer, and H.-P. Kriegel. The dc-tree: A fully dynamic index structure for data warehouses. In *ICDE*, pages 379–388, 2000.

[45] P. T. Eugster, P. A. Felber, R. Guerraoui, and A. M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[46] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *SIGMOD*, pages 115–126, 2001.

[47] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, pages 216–226, 1996.

[48] R. Fagin. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences*, 58(1):83–99, 1999.

[49] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[50] P. C. Fishburn. Preference structures and their numerical representations. *Theoretical Computer Science*, 217(2):359–383, 1999.

[51] P. Georgiadis, I. Kapantaidakis, V. Christophides, E. M. Nguer, and N. Spyratos. Efficient rewriting algorithms for preference queries. In *ICDE*, pages 1101–1110, 2008.

[52] U. Güntzer, W.-T. Balke, and W. Kiessling. Optimizing multi-feature queries for image databases. In *VLDB*, pages 419–428, 2000.

[53] U. Güntzer, W.-T. Balke, and W. Kiessling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, pages 622–628, 2001.

[54] L. Guo, S. Amer-Yahia, R. Ramakrishnan, J. Shanmugasundaram, U. Srivastava, and E. Vee. Efficient top-k processing over query-dependent functions. In *VLDB*, pages 1044–1055, 2008.

[55] B. Hafenrichter and W. Kiessling. Optimization of relational preference queries. In *ADC*, pages 175–184, 2005.

[56] S. O. Hansson. Preference logic. *Handbook of Philosophical Logic (D. Gabbay, Ed.)*, 8, 2001.

[57] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

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

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

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

[61] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, 2003.

[62] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD*, pages 259–270, 2001.

[63] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[64] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *VLDB J.*, 13(1):49–70, 2004.

[65] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB J.*, 13(3):207–221, 2004.

[66] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[67] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid. Rank-aware query optimization. In *SIGMOD*, pages 203–214, 2004.

[68] E. Jembere, M. O. Adigun, and S. S. Xulu. Mining context-based user preferences for m-services applications. In *Web Intelligence*, pages 757–763, 2007.

[69] B. Jiang, J. Pei, X. Lin, D. W. Cheung, and J. Han. Mining preferences from superior and inferior examples. In *KDD*, pages 390–398, 2008.

[70] T. Joachims. Optimizing search engines using clickthrough data. In *KDD*, 2002.

[71] S. Y. Jung, J.-H. Hong, and T.-S. Kim. A formal model for user preference. In *ICDM*, pages 235–242, 2002.

[72] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[73] M. G. Kendall. The treatment of ties in ranking problems. *Biometrika*, 33(3):239–251, 1945.

[74] W. Kiessling. Foundations of preferences in database systems. In *VLDB*, pages 311–322, 2002.

[75] W. Kiessling. Preference queries with sv-semantics. In *COMAD*, pages 15–26, 2005.

[76] W. Kiessling, B. Hafenrichter, S. Fischer, and S. Holland. Preference xpath: A query language for e-commerce. In *Wirtschaftsinformatik*, 2001.

[77] W. Kiessling and G. Köstler. Preference sql - design, implementation, experiences. In *VLDB*, pages 990–1001, 2002.

[78] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl. Grouplens: Applying collaborative filtering to usenet news. *Commun. ACM*, 40(3):77–87, 1997.

[79] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: an online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.

[80] G. Koutrika and Y. Ioannidis. Personalized queries under a generalized preference model. In *ICDE*, pages 841–852, 2005.

[81] G. Koutrika and Y. Ioannidis. Answering queries based on preference hierarchies. Technical Report 2008-6, Stanford InfoLab, 2008.

[82] G. Koutrika and Y. E. Ioannidis. Personalization of queries in database systems. In *ICDE*, pages 597–608, 2004.

[83] G. Koutrika and Y. E. Ioannidis. Constrained optimalities in query personalization. In *SIGMOD*, pages 73–84, 2005.

[84] M. Lacroix and P. Lavency. Preferences; putting more knowledge into queries. In *VLDB*, pages 217–225, 1987.

[85] C. Li, K. C.-C. Chang, and I. F. Ilyas. Supporting ad-hoc ranking aggregates. In *SIGMOD*, pages 61–72, 2006.

[86] Y. Li, Z. A. Bandar, and D. McLean. An approach for measuring semantic similarity between words using multiple information sources. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):871–882, 2003.

[87] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting stars: The k most representative skyline operator. In *ICDE*, pages 86–95, 2007.

[88] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, pages 563–574, 2006.

[89] H. Liu and H.-A. Jacobsen. Modeling uncertainties in publish/subscribe systems. In *ICDE*, pages 510–522, 2004.

[90] Y. Luo, X. Lin, W. W. 0011, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD*, pages 115–126, 2007.

[91] A. Machanavajjhala, E. Vee, M. N. Garofalakis, and J. Shanmugasundaram. Scalable ranked publish/subscribe. *PVLDB*, 1(1):451–462, 2008.

[92] A. Miele, E. Quintarelli, and L. Tanca. A methodology for preference-based personalization of contextual data. In *EDBT*, pages 287–298, 2009.

[93] G. A. Miller. Wordnet: a lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.

[94] B. Mobasher, R. Cooley, and J. Srivastava. Automatic personalization based on web usage mining. *Commun. ACM*, 43(8):142–151, 2000.

[95] R. J. Mooney and L. Roy. Content-based book recommending using learning for text categorization. *CoRR*, cs.DL/9902011, 1999.

[96] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, pages 281–290, 2001.

[97] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *Proceedings of the 15th International Conference on Data Engineering*, pages 22–29, 1999.

[98] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478, 2003.

[99] M. J. Pazzani and D. Billsus. Learning and revising user profiles: The identification of interesting web sites. *Machine Learning*, 27(3):313–331, 1997.

[100] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *VLDB*, pages 253–264, 2005.

[101] K. Pripuzic, I. P. Zarko, and K. Aberer. Top-k/w publish/subscribe: finding k most relevant publications in sliding time window w. In *DEBS*, pages 127–138, 2008.

[102] K. A. Ross, P. J. Stuckey, and A. Marian. Practical preference relations for large data sets. In *ICDE Workshops*, pages 229–236, 2007.

[103] A. Schmidt, A. K. Aidoo, A. Takaluoma, U. Tuomela, K. Laerhoven, and M. de Velde. Advanced interaction in context. In *1st Int'l Symposium on Handheld and Ubiquitous Computing*, pages 89–101, 1999.

[104] A. Simitsis, G. Koutrika, and Y. E. Ioannidis. Précis: from unstructured keywords as queries to structured databases as answers. *VLDB J.*, 17(1):117–149, 2008.

[105] D. Souravlias, M. Drosou, K. Stefanidis, and E. Pitoura. On novelty publish/subscribe delivery. *Submitted.*

[106] K. Stefanidis, M. Drosou, and E. Pitoura. Perk: Personalized keyword search in relational databases. *Submitted.*

[107] K. Stefanidis, M. Drosou, and E. Pitoura. You may also like results in relational databases. In *PersDB*, pages 37–42, 2009.

[108] K. Stefanidis, G. Koutrika, and E. Pitoura. A survey on representation, composition and application of preferences in database systems. *Submitted for journal publication.*

[109] K. Stefanidis and E. Pitoura. Approximate contextual preference scoring in digital libraries. In *PersDL*, pages 60–64, 2007.

[110] K. Stefanidis and E. Pitoura. Fast contextual preference scoring of database tuples. In *EDBT*, pages 344–355, 2008.

[111] K. Stefanidis, E. Pitoura, and P. Vassiliadis. Managing contextual preferences. *Submitted for journal publication.*

[112] K. Stefanidis, E. Pitoura, and P. Vassiliadis. Modeling and storing context-aware preferences. In *ADBIS*, pages 124–140, 2006.

[113] K. Stefanidis, E. Pitoura, and P. Vassiliadis. Adding context to preferences. In *ICDE*, pages 846–855, 2007.

[114] K. Stefanidis, E. Pitoura, and P. Vassiliadis. On relaxing contextual preference queries. In *MDM*, pages 289–293, 2007.

[115] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310, 2001.

[116] Y. Tao, X. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In *ICDE*, page 65, 2006.

[117] R. Torlone and P. Ciaccia. Finding the best when it's a matter of preference. In *SEBD*, pages 347–360, 2002.

[118] R. Torlone and P. Ciaccia. Management of user preferences in data intensive applications. In *SEBD*, 2003.

[119] A. H. van Bunningen, L. Feng, and P. M. G. Apers. A context-aware preference model for database querying in an ambient intelligent environment. In *DEXA*, pages 33–43, 2006.

[120] A. H. van Bunningen, M. M. Fokkinga, P. M. G. Apers, and L. Feng. Ranking query results using context-aware preferences. In *ICDE Workshops*, pages 269–276, 2007.

[121] P. Vassiliadis and S. Skiadopoulos. Modelling and optimisation issues for multidimensional databases. In *CAiSE*, pages 482–497, 2000.

[122] E. Vee, U. Srivastava, J. Shanmugasundaram, P. Bhat, and S. Amer-Yahia. Efficient computation of diverse query results. In *ICDE*, pages 228–236, 2008.

[123] Q. Wang, W.-T. Balke, W. Kiessling, and A. Huhn. P-news: Deeply personalized news dissemination for mpeg-7 based digital libraries. In *ECDL*, pages 256–268, 2004.

[124] M. P. Wellman and J. Doyle. Preferential semantics for goals. In *Proc. of AAAI*, pages 698–703, 1991.

[125] R. C.-W. Wong, J. Pei, A. W.-C. Fu, and K. Wang. Mining favorable facets. In *KDD*, pages 804–813, 2007.

[126] T. Xia, D. Zhang, and Y. Tao. On skylining with flexible dominance relation. In *ICDE*, pages 1397–1399, 2008.

[127] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *ICDE*, pages 189–200, 2003.

[128] G. You and S. Hwang. Search structures and algorithms for personalized ranking. *Information Sciences*, 178(20):3925–3942, 2008.

[129] C. Yu, L. V. S. Lakshmanan, and S. Amer-Yahia. It takes variety to make a world: diversification in recommender systems. In *EDBT*, 2009.

[130] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *VLDB*, pages 241–252, 2005.

[131] H. Zha, Z. Zheng, H. Fu, and G. Sun. Incorporating query difference for learning retrieval functions in world wide web search. In *CIKM*, pages 307–316, 2006.

[132] C. Zhai and J. D. Lafferty. A risk minimization framework for information retrieval. *Information Processing and Management*, 42(1):31–55, 2006.

[133] M. Zhang and N. Hurley. Avoiding monotony: improving the diversity of recommendation lists. In *RecSys*, 2008.

[134] M. Zhang and N. Hurley. Avoiding monotony: improving the diversity of recommendation lists. In *RecSys*, pages 123–130, 2008.

[135] X. Zhang and J. Chomicki. Profiling sets for preference querying. In *SEBD*, pages 34–44, 2008.

[136] C.-N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen. Improving recommendation lists through topic diversification. In *WWW*, 2005.

[137] C. Zimmer, C. Tryfonopoulos, K. Berberich, M. Koubarakis, and G. Weikum. Node behavior prediction for large-scale approximate information filtering. In *LSDS-IR*, 2007.

Publications

1. K. Stefanidis, E. Pitoura and P. Vassiliadis. Managing Contextual Preferences. *Submitted for journal publication* (2009).

2. K. Stefanidis, G. Koutrika and E. Pitoura. A Survey on Representation, Composition and Application of Preferences in Database Systems. *Submitted for journal publication* (2009).

3. K. Stefanidis, M. Drosou and E. Pitoura. PerK: Personalized Keyword Search in Relational Databases. 13th International Conference on *Extending Database Technology* (*EDBT* 2010). To appear.

4. D. Souravlias, M. Drosou, K. Stefanidis and E. Pitoura. On Novelty Publish/Subscribe Delivery. 4th International Workshop on Ranking in Databases (*DBRank* 2010), in conjunction with the ICDE 2010 Conference. To appear.

5. K. Stefanidis, M. Drosou and E. Pitoura. "You May Also Like" Results in Relational Databases. In Proc. of the 3rd International Workshop on *Personalized Access, Profile Management and Context Awareness in Databases* (*PersDB* 2009), in conjunction with the *VLDB* 2009 Conference, August 28, 2009, Lyon, France.

6. M. Drosou, K. Stefanidis and E. Pitoura. Preference-Aware Publish/Subscribe Delivery with Diversity. In Proc. of the 3rd International Conference on *Distributed Event-Based Systems* (*DEBS* 2009), July 6-9, 2009, Nashville, TN, USA.

7. M. Drosou, E. Pitoura and K. Stefanidis. Preferential Publish/Subscribe. In Proc. of the 2nd International Workshop on *Personalized Access, Profile Management and Context Awareness: Databases* (*PersDB* 2008), in conjunction with the *VLDB* 2008 Conference, August 23, 2008, Auckland, New Zealand.

8. K. Stefanidis and E. Pitoura. Fast Contextual Preference Scoring of Database Tuples. In Proc. of the 11th International Conference on *Extending Database Technology* (*EDBT* 2008), March 25-30, 2008, Nantes, France.

9. K. Stefanidis, E. Pitoura and P. Vassiliadis. A Context-Aware Preference Database System. International Journal of *Pervasive Computing and Communications*, vol. 3, no. 4, pp. 439-460, 2007. Emerald Group Publishing Limited.

10. K. Stefanidis and E. Pitoura. Approximate Contextual Preference Scoring in Digital Libraries. In Proc. of the 10th DELOS Thematic Workshop on *Personalized Access, Profile Management and Context Awareness in Digital Libraries* (*PersDL* 2007), in conjunction with the *UM* 2007 Conference, June 29-30, 2007, Corfu, Greece.

11. K. Stefanidis, E. Pitoura and P. Vassiliadis. On Relaxing Contextual Preference Queries. In Proc. of the 2nd International Workshop on *Managing Context Information and Semantics in Mobile Environments* (*MCISME* 2007), in conjunction with the *MDM* 2007 Conference, May 7, 2007, Mannheim, Germany.

12. K. Stefanidis, E. Pitoura and P. Vassiliadis. Adding Context to Preferences. In Proc. of the 23rd *International Conference on Data Engineering* (*ICDE* 2007), April 15-20, 2007, Istanbul, Turkey.

13. K. Stefanidis, E. Pitoura and P. Vassiliadis. Modeling and Storing Context-Aware Preferences. In Proc. of the 10th East-European Conference on *Advances in Databases and Information Systems* (*ADBIS* 2006), September 3-7, 2006, Thessaloniki, Greece.

14. K. Stefanidis, E. Pitoura and P. Vassiliadis. On Supporting Context-Aware Preferences in Relational Database Systems. In Proc. of the 1st International Workshop on *Managing Context Information in Mobile and Pervasive Environments* (*MCMP* 2005), in conjunction with the *MDM* 2005 Conference, May 9, 2005, Ayia Napa, Cyprus.

Short CV

Kostas Stefanidis was born on July 16, 1979 in Thessaloniki, Greece. He received his MSc and BSc Degrees from the Computer Science Department of the University of Ioannina in Greece in 2005 and 2003 respectively, both under the supervision of Prof. Evaggelia Pitoura. From 2002 until now, he is a member of the Distributed Data Management Laboratory. His research focus is on personalization systems, with particular interest in personalized search.