Διαχείριση Προτιμήσεων
που Βασίζονται σε Συμφραζόμενη Πληροφορία
σε Συστήματα Βάσεων Δεδομένων

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

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

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ
ΠΛΗΡΟΦΟΡΙΚΗ ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ
ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Οκτώβρης 2005

Acknowledgments

I would like to thank my supervisor Professor Evaggelia Pitoura for her help, support, the time spent during the elaboration of this thesis, and especially, the patience she has shown until this thesis is completed. Furthermore, I would like to thank Professor Panos Vassiliadis for sharing his knowledge with me, especially on the study of the OLAP techniques. I would also like to thank my friends Georgia Koloniari, George Rigas, Thodoris Tsotsos, and Andreas Fotiou for their useful contribution, and their help on many issues of this work.

Contents

Contents i

1 Introduction **5**

1.1 Context Preliminaries . . . 5
1.2 Scope of Thesis . . . 7
1.3 Thesis Outline . . . 9

2 A Logical Model for Context and Preferences **10**

2.1 Reference Example . . . 10
2.2 Modeling Context . . . 11
2.3 Contextual Preferences . . . 14
2.3.1 Basic Preferences . . . 14
2.3.2 Aggregate Preferences . . . 15
2.4 Inheriting Preferences . . . 15
2.5 Techniques for the Expression of Preferences . . . 17
2.6 Other Models of Preferences . . . 17

3 The Storage Model **21**

3.1 Storing Basic Preferences . . . 21
3.2 Storing Context Hierarchies . . . 22
3.3 Storing the Value Functions . . . 24
3.4 Storing Aggregate Preferences . . . 25

4 Querying Context **27**

4.1 Querying Simple Preferences . . . 27
4.2 Querying with Aggregate Scores . . . 28
4.3 Computing Aggregate Scores . . . 28
4.4 Traditional OLAP Operators . . . 29

5 Caching Context States **31**
5.1 The Context Tree 31
5.2 Querying the Context Tree 35
5.3 Querying with Approximate Results 36
5.4 Querying Using Hierarchies 40
5.5 Additional Issues 40
5.5.1 Replacement Policies 40
5.5.2 Mapping Context Parameters to Levels 41
5.6 Bloom-Based Index for the Context Tree 42
5.6.1 Bloom Filters Preliminaries 42
5.6.2 Multi-level Bloom Filters 43

6 Implementation and Evaluation **45**
6.1 Prototype Implementation 45
6.2 Performance Evaluation of the Context Tree 47
6.2.1 Evaluating the Size of the Context Tree 50
6.2.2 Evaluating the Accuracy of Approximate Results 51

7 Related Work **57**
7.1 Context-Awareness 57
7.2 Infrastructures for Context 58
7.3 Context-Aware DBMS 59
7.3.1 Context-Aware Query Processing 59
7.3.2 Architecture of Context-Aware DBMS 62
7.4 Context Management 62
7.4.1 Model of Context 63
7.4.2 Storing Context 65
7.4.3 Updating Context 65
7.5 Top-K Querying 66

8 Conclusions and Future Work **69**

List of Figures

2.1	The database schema of our running example.	11
2.2	Hierarchies on *location*.	12
2.3	The hierarchy tree for parameter L.	15
2.4	The hierarchy tree of location.	16
2.5	The entity relationship schema of Situated Preferences.	20
3.1	Data cubes for each context parameter.	22
3.2	The two fact tables of our schema (one for each context parameter) and the dimension tables for $Users$ and $Restaurants$.	23
3.3	A typical (left) and an extended dimension table (right).	24
3.4	A *cube* in the Context Relational model.	26
5.1	A context tree.	32
5.2	A set of aggregate preferences.	33
5.3	A context tree for a specific profile.	34
5.4	The hierarchy tree of *location*.	40
5.5	A Bloom filter with 4 hash functions.	42
5.6	The BBF for the context tree of Fig 5.3.	43
5.7	The DBF for the context tree of Fig 5.3.	44
6.1	Overall system architecture of a Context-Dependent Preference Database.	46
6.2	Query Example.	47
6.3	Result Example.	47
6.4	Result Example.	48
6.5	Zipf Data Distribution.	50
6.6	Uniform Data Distribution	51
6.7	Zipf Data Distribution with $a = 1.0$ (left) and $a = 1.5$ (right)	51
6.8	Different Results between two similar Queries.	52

6.9 Different Results between two similar Queries with or without similar Degrees. 53
6.10 Different Results between two similar Queries when $\varepsilon' = 0.04$, $\varepsilon' = 0.08$, and $\varepsilon' = 0.12$. 54
6.11 Different Results between two similar Queries when $\varepsilon' = 0.04$, $\varepsilon' = 0.08$, and $\varepsilon' = 0.12$. 54
6.12 Different Results between two similar Queries when $\varepsilon' = 0.04$, $\varepsilon' = 0.08$, and $\varepsilon' = 0.12$. 55
6.13 Different Results when the *Approximation Coverage Threshold* has the values 40%, 60%, 80%. 55
6.14 Different Results when the *Approximation Coverage Threshold* has values 40%, 60%, 80%. 56

7.1 Connecting Context and Databases 63

List of Tables

2.1 Preference SQL Queries 19

6.1 Input Parameters 49

List of Algorithms

1 Search_Path_Algorithm 36

Abstract

Konstantinos Stefanidis.
MSc, Computer Science Department, University of Ioannina, Greece.

October, 2005.
Context-Aware Preferences for Database Systems.
Supervisor: Evaggelia Pitoura.

A context-aware system is a system that uses context to provide relevant information or services to its users. While there has been a variety of context middleware infrastructures and context-aware applications, little work has been done on integrating context into database management systems. In this thesis, we consider a preference database system that facilitates context-aware queries, that is, queries whose result depends on the context at the time of their submission. At first, we present the fundamental concepts related to context modeling and define user preferences. We propose using data cubes to store the dependencies between context-dependent preferences and database relations and OLAP techniques for processing context-aware queries. This provides support for manipulating the captured context data at different levels of abstractions, for instance, in the case of a context parameter representing location, this allows us to express the fact that a city belongs to a country. To improve query performance, we use an auxiliary data structure, called context tree, to store precomputed preferences. A path in the context tree corresponds to an assignment of values to context parameters and thus to a context state. This tree provides an efficient way to retrieve *top-k* results that are relevant to a preference query. We further show how search in this data structure can be improved using an additional hash-based index to test for membership in the context tree. We also outline an implementation of a prototype application, and we evaluate the performance of the context tree on answering queries.

Περιληψη

Κωνσταντίνος Στεφανίδης του Χαραλάμπους και της Ευτυχίας.
MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Οκτώβρης, 2005.
Διαχείριση Προτιμήσεων που Βασίζονται σε Συμφραζόμενη Πληροφορία σε Συστήματα Βάσεων Δεδομένων.
Επιβλέπουσα: Ευαγγελία Πιτουρά.

Context είναι οποιαδήποτε πληροφορία μπορεί να χρησιμοποιηθεί για να χαρακτηρίσει μία κατάσταση ή μία οντότητα. Μία οντότητα είναι ένα άτομο, μία τοποθεσία ή ένα αντικείμενο που μπορεί να θεωρηθεί σχετικό με την αλληλεπίδραση ενός χρήστη και μίας εφαρμογής, συμπεριλαμβανομένου του χρήστη και της εφαρμογής. Προκειμένου ένα σύστημα να παρέχει καλύτερες υπηρεσίες στους χρήστες απαιτείται να είναι ενήμερο για τις παραμέτρους που επηρεάζουν την εκτέλεση των υπηρεσιών. Οι παράμετροι αυτοί αποτελούν το context του συστήματος. Κατ΄ αυτόν τον τρόπο, ένα σύστημα είναι *context-aware* αν χρησιμοποιεί το context για να παρέχει στους χρήστες σχετικές πληροφορίες και υπηρεσίες, προσαρμόζοντας κατάλληλα τη λειτουργία του.

Παρά το γεγονός ότι το context είναι μία γενική έννοια, μία κατηγοριοποίησή του είναι η παρακάτω:

- Το *υπολογιστικό context* που αναφέρεται στο κόστος επικοινωνίας, στη δικτυακή σύνδεση, σε πόρους που βρίσκονται σε κοντινή απόσταση (όπως εκτυπωτές), κ.α.
- Το *context χρήστη* που περιλαμβάνει τις προτιμήσεις του χρήστη, τη θέση του, τα πρόσωπα δίπλα του, κ.α.
- Το *φυσικό context* που αναφέρεται σε παράγοντες του φυσικού περιβάλλοντος όπως θερμοκρασία, επίπεδα θορύβου, κ.α.
- Ο *χρόνος* που μπορεί να δηλώνεται ως μία χρονική στιγμή μέσα στη μέρα, ως μήνας ή εποχή, κτλ.

Παρόλο που υπάρχουν διάφορες context-aware εφαρμογές, λίγη σχετική δουλειά έχει γίνει στην ενσωμάτωση του context σε συστήματα διαχείρισης βάσεων δεδομένων. Σε αυτήν τη μεταπτυχιακή εργασία εξειδίκευσης ορίζουμε ένα σύστημα βάσεων δεδομένων με προτιμήσεις, το οποίο υποστηρίζει *context-aware ερωτήσεις*. Το αποτέλεσμα των ερωτήσεων αυτών εξαρτάται από τις τιμές που έχουν οι παράμετροι του context τη στιγμή που εκφράζεται η ερώτηση.

Αρχικά, παρουσιάζουμε πως μοντελοποιείται το context και στη συνέχεια ορίζουμε τις προτιμήσεις των χρηστών. Ένας χρήστης εκφράζει την προτίμηση του για ένα δεδομένο δίνοντας ένα βαθμό ενδιαφέροντος, δηλαδή έναν αριθμό μεταξύ του 0 και του 1. Διακρίνουμε τις προτιμήσεις σε βασικές και σύνθετες. Μία βασική προτίμηση περιγράφεται από μία μόνο παράμετρο του context, ενώ μία σύνθετη από όλες. Ο βαθμός ενδιαφέροντος για μία σύνθετη προτίμηση υπολογίζεται από τους επιμέρους βαθμούς των αντίστοιχων βασικών προτιμήσεων χρησιμοποιώντας μία γραμμική συνάρτηση. Επιπλέον, αν μία παράμετρος του context συμμετέχει σε διαφορετικά επίπεδα μίας ιεραρχίας είναι δυνατό μία προτίμηση να εκφραστεί σε παραπάνω του ενός επίπεδα.

Προτείνουμε τη χρήση υπερκύβων δεδομένων για την αποθήκευση των προτιμήσεων. Για την ακρίβεια αποθηκεύονται σε κύβους μόνο οι βασικές προτιμήσεις των χρηστών. Ο αριθμός των κύβων είναι ίσος με τον αριθμό των παραμέτρων του context, καθώς υπάρχει ένας κύβος για κάθε παράμετρο. Σε κάθε κύβο υπάρχει μία διάσταση για την παράμετρο του context, ενώ οι υπόλοιπες εκφράζουν γνωρίσματα που δεν σχετίζονται με το context. Σε κάθε κελί του κύβου αποθηκεύεται ο βαθμός ενδιαφέροντος της αντίστοιχης προτίμησης. Ένα πλεονέκτημα της χρήσης των κύβων για την αποθήκευση των προτιμήσεων του χρήστη είναι ότι μας παρέχουν τη δυνατότητα να χειριστούμε τα δεδομένα σε διαφορετικά επίπεδα ιεραρχίας. Έτσι για παράδειγμα, αν η γεωγραφική θέση ενός χρήστη αποτελεί παράμετρο του context, αυτό μας επιτρέπει την έκφραση μίας προτίμησης σε επίπεδο περιοχής ή σε επίπεδο πόλης. Οι σύνθετες προτιμήσεις δεν αποθηκεύονται σε κύβους με σκοπό τη βελτίωση της απόδοσης του συστήματος σε χρόνο και σε χώρο. Αποθηκεύονται μόνο οι προτιμήσεις που έχουν υπολογιστεί από προηγούμενες ερωτήσεις σε ένα δέντρο που ονομάζεται context tree και χρησιμοποιείται ως ευρετήριο (αναφέρεται παρακάτω).

Επιπλέον, παρουσιάζουμε ένα σύνολο από διαφορετικά είδη ερωτήσεων, που μπορούν να τεθούν στο σύστημά μας. Οι ερωτήσεις αυτές μπορούν να απαντηθούν χρησιμοποιώντας πληροφορία που βασίζεται στο συνδυασμό των προτιμήσεων και του context. Επεξηγούμε τους τελεστές slice, dice, roll-up, και drill-down της OLAP, καθώς και αναφέρουμε πως χρησιμοποιούνται οι παραπάνω τελεστές στην εκτέλεση των ερωτήσεων.

Στη συνέχεια, αναλύουμε διάφορα θέματα που αφορούν τη βελτίωση της απόδοσης εκτέλεσης ερωτήσεων. Πιο συγκεκριμένα, περιγράφουμε μία δομή δεδομένων

που ονομάζεται *context tree*. Χρησιμοποιούμε τη δομή αυτή για την αποθήκευση των αποτελεσμάτων προηγούμενων ερωτήσεων (για συγκεκριμένες τιμές των παραμέτρων του context), ώστε αυτά τα αποτελέσματα να χρησιμοποιηθούν από επόμενες ερωτήσεις. Σε κάθε επίπεδο του δέντρου αναπαριστάται μία παράμετρος του context. Επομένως, ένα μονοπάτι του δέντρου αντιστοιχεί σε μία ανάθεση τιμών σε κάθε παράμετρο. Κάθε κόμβος του δέντρου (εκτός των φύλλων) περιέχει ένα σύνολο από ζεύγη της μορφής [key, pointer], ενώ τα φύλλα περιέχουν τα k πρώτα δεδομένα με το υψηλότερο συνολικό βαθμό ενδιαφέροντος.

Το context tree μας παρέχει έναν αποδοτικό τρόπο για την ανάκτηση των k καλύτερων αποτελεσμάτων μίας ερώτησης. Όταν μία ερώτηση τεθεί στο σύστημα, αρχικά αναζητούμε αν υπάρχει το αντίστοιχο μονοπάτι στο δεντρο. Αν ναι, ανακτούμε τα k καλύτερα αποτελέσματα από το αντίστοιχο φύλλο του δέντρου. Διαφορετικά, υπολογίζουμε την ερώτηση, και εισάγουμε στο δέντρο ένα νέο μονοπάτι (το αντίστοιχο της ερώτησης) μαζί με τα k καλύτερα αποτελέσματα. Επιπρόσθετα, επεκτείνουμε το μοντέλο ώστε να υποστηρίζει ερωτήσεις χωρίς να απαιτείται ο υπολογισμός των k καλύτερων αποτελεσμάτων από την αρχή. Τούτο συμβαίνει όταν υπάρχουν αποθηκευμένες παρόμοιες ερωτήσεις στο context tree.

Η απόδοση αναζήτησης μονοπατιού στη δομή δεδομένων που προαναφέρθηκε μπορεί να βελτιωθεί χρησιμοποιώντας επιπρόσθετα ευρετήρια που βασίζονται στον κατακερματισμό και ονομάζονται Bloom φίλτρα. Τα Bloom φίλτρα υποστηρίζουν αποδοτικά ερωτήσεις που αναφέρονται στην παρουσία ή όχι ενός αντικειμένου. Έτσι, όταν τίθεται στο σύστημα μία νέα ερώτηση, που αντιστοιχεί σε συγκεκριμένη ανάθεση τιμών για τις παραμέτρους του context, αντί να αναζητήσουμε το κατάλληλο μονοπάτι στο context tree, χρησιμοποιούμε πρώτα τα Bloom φίλτρα. Αυτά μας παρέχουν μία γρήγορη απάντηση για το αν υπάρχει ή όχι το μονοπάτι στο δέντρο. Επομένως, αν δεν υπάρχει, αποφεύγουμε την αναζήτηση στο context tree.

Επιπρόσθετα, υλοποιήσαμε μία εφαρμογή, στην οποία επιτρέπεται ένας χρήστης να εκφράσει τις προτιμήσεις του για ένα σύστημα βάσεων δεδομένων σχετικό με εστιατόρια. Οι προτιμήσεις αυτές σχετίζονται με δύο παραμέτρους context: τον καιρό και τη θέση-τοποθεσία του χρήστη. Οι χρήστες έχουν τη δυνατότητα να θέτουν ερωτήσεις στο σύστημα, ενώ οι απαντήσεις που λαμβάνουν εξαρτώνται από τις τρέχουσες τιμές του context. Τέλος, αποτιμήσαμε την απόδοση του context tree, εκτελώντας μία σειρά από πειράματα.

Chapter 1

Introduction

1.1 Context Preliminaries

1.2 Scope of Thesis

1.3 Thesis Outline

1.1 Context Preliminaries

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 the application themselves [1]. There are various types of context such as time, location, computing devices and user's profiles.

While context is a general term, a classification of the most common types of context used in building software systems is the following:

- *Computing context*. Computing context includes (i) network connectivity, communication costs and communication bandwidth, (ii) nearby resources such as printers, displays and workstations and (iii) local resources such as cpu, energy and type of display. Such parameters affect data engineering since to improve performance data engineering mechanisms (such as query processing algorithms and concurrency control protocols) must take into account the underlying resources.

- *User context*. User context includes the user's profile, location, people nearby, even the current social situation. This type of context parameters directly affect the type of information relevant to a user. Thus, the user context affects the results of query processing.

- *Physical context*. Physical context refers to the environment surrounding the user such as lighting, noise levels, traffic conditions and temperature. Such type of context indirectly affects the type of information that is relevant for, or interesting to the user.

- *Time context*. Time context refers to the typical characterizations of time such as time of a day, week, month and season of the year. Time may affect the result of a query, since relevance may also be dependent on the time.

The above taxonomy of context types is introduced in [2]. Each type is called *context parameter*. There are dependencies among the various types of context parameters. For instance, time context may affect the computing context, e.g., network traffic at weekends is less than during weekdays.

Context information often involves real world entities. Thus, it makes sense to measure the quality of context information (QoI), or the extent to which the data corresponds to the real world. The quality of context information can vary, perhaps substantially, depending on the context source and the type of context. The following characteristics of context necessitate the introduction of appropriate metrics for assessing the quality of context information. Such characteristics include the following [3]:

- Context exhibits a range of temporal characteristics. Some types of context (such as the user profiles) are relatively *static* whereas other types of context (such as location) are *dynamic*. Furthermore, storing the context history is important for predicting future values of context and developing appropriate models for context evolution.

- Context information is imperfect. There are various reasons for that. One reason stems from the fact that context often are dynamic, thus it gets quickly out-dated. Then, some forms of context information is often produced by crude sensor inputs which may result in faulty information. Furthermore, due to disconnections (e.g., wireless connections becoming unavailable) or failures (e.g., sensors running out of battery power), the available information may be imprecise. Finally, there is often the need for time-consuming transformations for producing usable values of context. Often the overhead of such transformations may be avoided or reduced on the cost of producing rougher estimations.

- There are many alternative presentations of context offering varying details and depth.
- Finally, context parameters are highly interrelated. There are complex dependencies among them that are sometimes difficult to deduce. Such dependencies may lead to conflicting and sometimes inconsistent results.

There are many issues regarding the quality of context information. First, one must identify what are the parameters that characterize quality. Then, one must also determine how are these parameters measured, that is, what is an appropriate metric for each one of the quality parameters. Furthermore, an important issue is the estimation of quality provided and more importantly how is this quality guaranteed. Research has not sufficiently addressed all these issues. Most comprehensive research efforts regarding handling the quality of context focus on a particular type of context, that of *location*. Various models have been developed for predicting the location of moving objects. A survey on location management can be found in [4].

In general, we identify the following *Quality of Service (QoS)* parameters as relevant to context information and data management:

- *Accuracy*: the deviation of the estimated value of a context parameter from the actual value of the parameter
- *Level of Detail*: this refers to the granularity of the presented information, for instance, in the case of time, this can be measured e.g., in years, hours or seconds
- *Conflict-free*: this measure is application-dependent and refers to the requirement of having consistent and non-contradicting values
- *Timeliness*: the deviation of the value of the context parameter in time (the extent to which a value is kept up-to-date)

A *context-aware* system is a system that uses context to provide relevant information and/or services to its users, where relevancy depends on the users' task. Although there has been a lot of work on developing a variety of context infrastructures and context-aware middleware and applications (for example, the Context Toolkit [5] and the Dartmouth Solar System [6]), there has been little work concerning the integration of context information into databases.

1.2 Scope of Thesis

In this thesis, we investigate the use of context in relational database management systems. We consider *context-aware* queries that are queries whose results

depend on the context at the time of their submission. In particular, users express their preferences on specific attributes of a relation. Such preferences depend on context, that is, they may have different values depending on context.

We model context as a finite set of special-purpose attributes, called *context parameters*. Examples of context parameters are location, weather and the type of computing device in use. A context state is an assignment of values to context parameters. Users express their preferences on specific database instances based on a single context parameter. Such *basic preferences*, i.e., preferences associating database relations with a single context attribute, are combined to compute *aggregate preferences* that include more than one context parameter.

We store basic preferences in data cubes, following the OLAP paradigm. An advantage of using cubes and OLAP techniques is that they provide the capability of using hierarchies to introduce different levels of abstraction for the captured context data. For instance, this allows us to aggregate data along the location context parameter, by grouping preferences for all cities of a specific country. We show how context-aware preference queries are processed and the role of OLAP techniques in their manipulation.

Aggregate preferences are not explicitly stored. To improve performance, we propose storing aggregate preferences computed as results of previous queries using an auxiliary data structure called *context tree*. A path in the context tree corresponds to an assignment of values to context parameters, that is, to a context state, for which the aggregate score has been previously computed.

The context tree provides an efficient way to retrieve the *top-k* results that are relevant to a preference query. When a query is posed to the system, we first check if there exists a context state that matches it in the context tree. If so, we retrieve the *top-k* results from the associated leaf node. Otherwise, we compute the answer and insert the new context state, i.e., the new path and the associated *top-k* results, in the tree. Furthermore, the results stored in the context tree can be re-used to speed-up query processing. This happens for a similar query, i.e., for a query whose context state has similar values with a previous one. In this case, the results of the new query are approximate. We also show how search in the context tree can be improved by using a variation of a Bloom-based filter for testing membership in the tree.

As a proof-of-concept, we have implemented a simple application that allows users to express their preferences regarding a restaurant database. These preferences depend on two context parameters, location and weather. Then, users can pose preference queries whose results depend on context. Furthermore, we evaluate the performance of the context tree on answering queries, running a set of experiments.

Summarizing, we make the following contributions:

- We provide a logical model for the representation of user preferences and context-related information. The impact of context information on the evaluation of user preferences is explicitly traced.
- We discuss the implementation of our model in a relational DBMS.
- We investigate the usage of On-Line Analytical Processing (OLAP) techniques for the manipulation of context-aware query operations.
- We propose storing previously computed aggregate scores using a data structure termed *context tree* that indexes these results based on the context parameters.

1.3 Thesis Outline

Chapter 2 describes our reference example, which is used in the rest of the thesis to explain our approach. Furthermore, in the same chapter we introduce our preference model. Chapter 3 focuses on how preferences are stored. Chapter 4 discusses query processing in our framework. Chapter 5 introduces the context tree for storing aggregate preferences. Our prototype implementation is outlined in Chapter 6. In the same chapter, we present the performance evaluation of the context tree, while related work is presented in Chapter 7. Chapter 8 concludes this thesis with a summary of our contributions and directions for future work.

Chapter 2

A Logical Model for Context and Preferences

2.1 Reference Example

2.2 Modeling Context

2.3 Contextual Preferences

2.4 Inheriting Preferences

2.5 On Expressing Preferences

2.6 Other Models of Preferences

Our model is based on relating context and database relations through preferences. In this chapter, we first introduce a reference example used to explain our approach, throughout this thesis. Then, we present the fundamental concepts related to context modeling and define user preferences.

2.1 Reference Example

Consider a database schema with information about restaurants and users (Fig. 2.1). In this application, we consider two context parameters as relevant: *location* and *weather*. Users have preferences about restaurants that they express by providing a numeric score between 0 and 1. The degree of interest that a

Restaurant(rid, name, phone, region, cuisine)
User(uid, name, phone, address, e-mail)

Figure 2.1: The database schema of our running example.

user expresses for a restaurant depends on the values of the context parameters. For instance, a user may want to eat different kinds of food when the weather is *rainy, cloudy* or *sunshine*. For example, user $Mary$ may give to restaurant *Zoloushka* that serves "Russian" food a higher score when the weather is *rainy* than when the weather is $sunshine$. Furthermore, the current user's location affects the result of a query, for example, a user may prefer restaurants that are nearby her current location. Thus, a user's preference on a specific restaurant depends on the context parameters. A user can specify preferences without giving values for all context parameters, i.e., *preference(187, 334, *, rainy) = 0.8* means that the restaurant *Zoloushka* with $id = 187$, for user $Mary$ with $id = 334$ has *interest score 0.8*, when the weather is *rainy*, independently of the user's location. In general, when a context parameter has the special value $*$, any value is acceptable.

2.2 Modeling Context

The modeling of context relies on several fundamental concepts. As usual, domains represent the available types and collections of values of the system. Context parameters refer to the available set of attributes that the database designer will chose to represent context. At any point in time, a context state refers to an instantiation of the context parameters at this point. Context parameters are extended with OLAP-like hierarchies, in order to enable a richer set of query operations to be applied over them.

Domains. A *domain* is an infinitely countable set of values. All domains are enriched with a special value for representing NULL, the semantics of which refers to our lack of knowledge.

Attributes and Relations. As usual, we assume a countable collection of attribute names. Each attribute A_i is characterized by a name and a domain $dom(A_i)$. A relation schema is a finite set of attributes and a relation instance is a finite subset of the Cartesian product of the domains of the relation schema.

Context Parameters. Context is modeled through a finite set of special-purpose attributes, called *context parameters* (c_i). For a given application X,

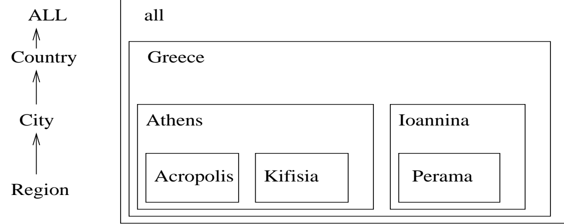

Figure 2.2: Hierarchies on *location*.

we can define its context environment C_X as a set of n context parameters $\{c_1, c_2, \ldots, c_n\}$.

Context State. In general, a *context state* is an assignment of values to context parameters. The context state at time instant t is a tuple with the values of the context parameters at time instant t, $CS_X(t) = \{c_1(t), c_2(t), \ldots c_n(t)\}$, where $c_i(t)$ is the value of the context parameter c_i at timepoint t. For instance, assuming *location* and *weather* as context parameters, a context state can be: $CS(current) = \{Acropolis, sunshine\}$.

Hierarchies for Attributes. It is possible for an attribute to participate in an associated *hierarchy of levels* of aggregated data i.e., it can be viewed from different levels of detail. Formally, an *attribute hierarchy* is a lattice of attributes – called *levels* for the purpose of the hierarchy – $L = (L_1, \ldots, L_n, ALL)$. We require that the upper bound of the lattice is always the level ALL, so that we can group all the values into the single value $'all'$. The lower bound of the lattice is called the detailed level of the parameter. For instance, let us consider the hierarchy *location* of Fig. 2.2. Levels of *location* are *Region*, *City*, *Country*, and ALL. *Region* is the most detailed level. Level ALL is the most coarse level for all the levels of a hierarchy. Aggregating to the level ALL of a hierarchy ignores the respective parameter in the grouping (i.e., practically groups the data with respect to all the other parameters, except for this particular one).

The relationship between the values of the context levels is achieved through the use of the set of $anc_{L_1}^{L_2}$ functions. A function $anc_{L_1}^{L_2}$ assigns a value of the domain of L_2 to a value of the domain of L_1. For instance, $anc_{Region}^{City}(Acropolis) = Athens$. A formal definition of these hierarchies can be found in [7].

Dynamic and Static Context Parameters. Traditionally, data stored in databases are considered constant unless it is explicitly modified. Since, some context values change continuously with time, we represent the corresponding

context parameters as functions of time. In that way, using functions of time we can compute the value of a parameter at the point we want to use it, without needing continuously updates. Related work has been done in the context of managing the location of moving objects in [8, 9]. In our work, we discriminate between two kinds of context parameters: (a) static and (b) dynamic context parameters. *Static context parameters* take as value a simple value out of their domain. *Dynamic context parameters* on the other hand, are instantiated by the application of a function, the result of which is an instance of the domain of the context parameter.

Thus, we can represent context values as functions of time; context changes as time passes, even without an explicit update. For example, the distance between $Mary$ and a restaurant that she wants to visit is given as a function of time. This distance continually changes, when $Mary$ is moving. In that way, the sorted result of a query might be different, when the same query is entered in the system at several times, even if the time interval between them is very small. The explanation is that after a few minutes the user might be closer to another restaurant. From the above, we consider that the answer to a query depends not only on the database content, but also on the time at which the query is executed.

In such a model, we can enter queries that refer to past time or to the future. The answer to future queries in a usual database is frequently tentative. Suppose that $Mary$ is moving with her car. In the system is entered the query Q=*"retrieve the position of the car, after an hour"*. When we represent the position of the car as a function of time, we can have the prospective knowledge to estimate the position in a future time, if we know for example the speed and the direction of the car. So, there is no need to update continuously the position of the car, because we can compute its new one. Also, we can answer queries such as Q, i.e., queries that referred at future or at past time.

When a value of a context parameter is a function of time, it changes over time according to some given equation, even if it is not explicitly updated. In contrast, if the value is stored in the database in the traditional sense, it changes only when an explicit update occurs. For example, we can compute the position of $Mary's$ car ($Mary's$ location) when we know the time interval, the speed and the direction of the car. More specifically, when we know the previous location of the car, the current car's speed, its direction and the time interval between the two positions, we can compute the current position of the car. With this technique, it is not necessary to continuously update (explicitly) the car's position, but modify the speed and the direction of the car, when these change.

In our example, we assume that *weather* is a static parameter, i.e., each new value for *weather* is derived by an explicit update. On the other hand,

location is a dynamic parameter. In particular, *location* is defined as a function of time and in that way, we can compute the value of this parameter at the point we want to use it, without the need for continuous explicit updates. Defining appropriate functions and procedures for determining the value of a dynamic context parameter in the current or some future time instant is beyond the scope of this thesis.

2.3 Contextual Preferences

In this section, we define how a context state affects the results of a query. In our model, each user expresses his/her preference by providing a numeric score between 0 and 1 [10]. This score expresses a degree of interest, which is a real number. Value 1 indicates extreme interest. In reverse, value 0 indicates no interest for a preference. The special value $\oslash$ for a preference, means that there is a user's veto for the preference. Furthermore, the value $*$ represents that any value is acceptable.

More specifically, we divide preferences into basic (concerning a single context parameter) and aggregate ones (concerning a combination of context parameters).

2.3.1 Basic Preferences

Each basic preference is described by (a) a context parameter c_i, (b) a set of non-context parameters A_i, and (c) a degree of interest, i.e., a real number between 0 and 1. So, for the context parameter c_i, we have:

$$preference_{basic_i}(c_i, A_{k+1}, \ldots, A_n) = interest_score_i$$

In our reference example, there are two context parameters, *location* and *weather*. Also, the set of non-context parameters are attributes about *restaurants* and *users* (in this case the user is $Mary$) that are stored in the database. From $Mary's$ profile, we know that when she is at *Acropolis* she gives at the restaurant *BeauBrummel* the score 0.8, and when the weather is *cloudy* the same restaurant has score 0.9. In order to explain $Mary's$ high scores to the above preferences, we refer that the restaurant *BeauBrummel* is located in *Athens*, near *Acropolis*, and $Mary$ likes to eat $french$ cuisine when the weather is *cloudy* (*BeauBrummel* has $french$ cuisine). So, the basic preferences are:

$$preference_{basic_1}(Acropolis, BeauBrummel, Mary) = 0.8,$$
$$preference_{basic_2}(cloudy, BeauBrummel, Mary) = 0.9.$$

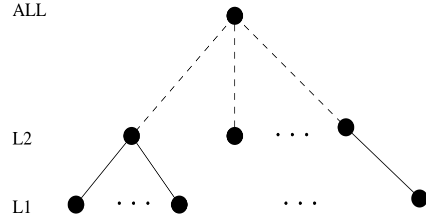

Figure 2.3: The hierarchy tree for parameter L.

2.3.2 Aggregate Preferences

Each aggregate preference is derived from a combination of basic preferences. The aggregate preference is expressed by a set of context parameters c_i and a set of non-context parameters A_i, and has a degree of interest

$$(preference(c_1, \ldots c_k, A_{k+1}, \ldots, A_n) = interest_score).$$

The interest score of the aggregate preference is a *value function* of the individuals scores (the degrees of the basic preferences). The value function prescribes how to combine basic preferences to produce the aggregate score, according to the user's profile. Users define in their profile how the basic scores contribute to the aggregate, giving a weight to each context parameter. So, if the weight for a context parameter is w_i the interest score will be: $interest_score =$

$$w_1 * interest_score_1 + \ldots + w_k * interest_score_k.$$

In the previous example if the weight of *location* is 0.6 and the weight of *weather* is 0.4, the preference has score: $0.6 * 0.8 + 0.4 * 0.9 = 0.84$ (from the above value function). Thus, we have:

$$preference(Acropolis, cloudy, BeauBrummel, Mary) = 0.84.$$

Note also, that when a basic preference has a *veto* ($\oslash$) value, this preference does not contribute to the creation of the aggregate one.

2.4 Inheriting Preferences

When the context parameter of a basic preference participates in different levels of a hierarchy, users can express their preference in any level, as well in more than one level. For example, $Mary$ can denote that the restaurant *Beau Brummel* has interest score 0.8 when she is at $Kifisia$ and 0.6 when she is in *Athens*. Note that in the hierarchy of location the city of *Athens* is one level up the region of $Kifisia$.

The tree of Fig. 2.3 represents the different levels of hierarchy for a context parameter. For the parameter L, let L_1, L_2,..., L_n, ALL be the different levels

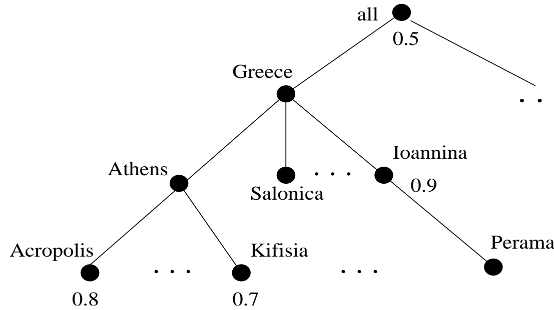

Figure 2.4: The hierarchy tree of location.

of the hierarchy, which can take various different values. There is a hierarchy tree, for each combination of non-context parameters. In our reference example (Fig. 2.4), there is a hierarchy tree for each user profile and for a specific restaurant that represents the interest scores of the user for the restaurants, accordingly to the context parameter's hierarchy. The root of the tree concerns level *ALL* with the single value *all*. The values of a certain dimension level L are found in the same level of the tree (e.g., *Athens* and *Salonica*, being both members of the dimension level *City*, are found at the same level of the tree in Fig. 2.4). The ancestor relationships $anc_{L_1}^{L_2}$ are translated to parent-child relationships in the tree (e.g., the node *Greece* is the parent of the node *Athens*). Each node is characterized by a score value for the preference of concerning the combination of the non-context attributes with the context value of the node.

If the query conditions refer to a level of the tree in which there is no explicit score given by the user, we propose three ways to find the appropriate score for a preference. In the first approach, we traverse the tree upwards until we find the first predecessor for which a score is specified. In this case, we assume that a user that defines a score for a specific level, implicitly defines the same score for all the following levels. In the second one, we compute the average score of all the successors of the immediately following level. This approach has a main drawback. When the successors have no score, we must execute the first method. Finally, in a more sophisticated approach we can compute a weighted average score combining the scores from both the predecessor and the successors.

In any of the above cases, if no score is defined at any level of the hierarchy, there is a default score of 0.5. Furthermore, it is possible for a score to exist at a certain level, but the query is referred to a higher level. In these circumstances, if the first or the third approach is chosen, and there is no score for any predecessor, we use the default score 0.5.

Take for example, Fig. 2.4 that depicts a hierarchy for a *user* and a

restaurant. So, for instance the restaurant *Beau Brummel* has score 0.8 when *Mary* is near *Acropolis*, 0.7 when she is in *Kifisia*, and 0.9 when she is in *Ioannina*. The root of the hierarchy has the default score 0.5. These degrees of interest scores, except the last one, have been explicitly defined by the user at her profile. If the query conditions refer to *Athens*, for which there is no score, the first approach gives score 0.5, because this is the first predecessor's score. If we choose the second approach, this leads to score $(0.8 + 0.7)/2 = 0.75$, while the third one produces a weighted combination of the above scores.

2.5 Techniques for the Expression of Preferences

To facilitate the procedure of expressing interests, the system may provide sets of pre-specified profiles with specific context-dependent preference values for the non-context parameters as well as default weights for computing the aggregate scores. In this case, instead of explicitly specifying basic and aggregate preferences for the non-context parameters, users may just select the profile that best matches their interests from the set of the available ones. By doing so, the user adopts the preferences specified by the selected profile.

Since the focus of this work is on efficiently combining preferences and database operations, our working assumption is that preferences are explicitly specified by users. Alternatively, preferences may be deduced by the previous behavior of the user, for instance by using data mining techniques on the history of the user database accesses. The issue of implicitly inferring preferences is orthogonal to the work presented in this thesis. There has been some previous work on the topic [11], that can be integrated in our approach.

2.6 Other Models of Preferences

The research literature on preferences is extensive. In particular, in the context of database queries, there are two different approaches for expressing preferences: a *quantitative* and a *qualitative* one. Both the quantitative and the qualitative approaches can be integrated with query processing.

With the *quantitative approach*, preferences are expressed indirectly by using scoring functions that associate a numeric score with every tuple of the query answer. Such a general quantitative framework for expressing and combining preferences is proposed in [10]. In this framework, a preference is expressed by the user for an entity. Entities are described by record types which are sets of named fields, where each field can take values from a certain type. The '*' symbol is used to match any element of that type. Preferences are ex-

pressed as functions that map entities of a given record type to a numerical score. This is the main difference with our approach, because we assign explicitly a numerical score to the corresponding entities. A set of preferences can be combined using a generic combine operator which is instantiated with a value function. For example, the preference of a user for restaurants can be expressed as $preference(type_of_food)$, with values $preference(chinese) = 0.1$, $preference(greek) = 0.8$ and $preference(other) = 0.1$.

In the quantitative framework of [12], user preferences are stored as degrees of interests in *atomic query elements* (such as individual selection or join conditions). The degree of interest expresses the interest of a person to include the associated condition into the qualification of a given query. Specific rules are specified for deriving preference of complex queries by building on stored atomic ones. The results of a query are ranked based on the estimated degree of interest in the combination of preferences they satisfy. Our approach can be generalized for this framework as well, either by including contextual parameters in the atomic query elements or by making the degree of interest for each atomic query element depend on context.

Both quantitative frameworks can be readily extended to include context. One way this can be achieved is by defining preference functions based on context. Then, based on the current values of context, the associated preference functions can be selected, combined and used to rank the results of any given query. Similarly, we may either include contextual parameters in the atomic query elements or make the degree of interest for each atomic query element depend on context.

In the *qualitative approach*, the preferences between the tuples in the answer to a query are specified directly, typically using binary preference relations. For example, one may express that $restaurant_1$ is preferred from $restaurant_2$ if their opening hours are the same and its price is lower. This framework can also be readily extended to include context. For instance, one may express that $restaurant_1$ is preferred from $restaurant_2$ if their opening hours are the same, its price is lower and it is closer to the current user's location. A logical qualitative framework is presented in [13] for formulating preferences as *preference formulas*. The preference formula is a first-order formula defining a preference relation between two tuples.

A different approach to expressing preferences is presented in [14, 15]. This approach introduces Preference SQL that extends SQL with a preference model. In standard SQL, database queries are characterized by hard constraints, in the sense that user wishes are either satisfied completely or not at all (any preference is translated in the *where* clause). Preference SQL provides a paradigm shift from the exact matches towards a best possible match-making, i.e., preferences

Table 2.1: Preference SQL Queries

Q_1	*select* * *from* [table name] *preferring* [attribute name] *around*[value];
Q_2	*select* * *from* [table name] *preferring* *highest*([attribute name]);

are to be treated as soft constraints. That means finding the best possible match between one's wishes and the reality.

Preference SQL includes a variety of built-in base preference types. In particular, there is a number of different selection criteria that refer to approximation, minimization, maximization, favorites, dislikes, etc. For example, observe the queries in Table 2.1. The first query (Q_1) uses the preference type *around* to express a positive preference to values close to a numerical target value. The second query (Q_2) asks for the highest value, if it is possible. Otherwise, the closest value to the maximum is considered acceptable.

Many times a "wish" cannot be expressed by a basic preference solely. For this cases, Preference SQL offers complex preferences that are based on a combination of basic ones. For instance, the *pareto accumulation* is the 'AND'-ing of basic preferences. Imagine that a customer buying a computer considers a maximum memory size and CPU speed as equally important. This preference can be expressed as:

- *select* * *from* computers
  preferring *highest*(main_memory) and
  highest(cpu_speed);

Our approach differs from Preference SQL on how preferences are expressed. In Preference SQL, preferences are implemented using operators and the result shows that an object is more preferable than another, while in our approach the results are ordered according to the degrees of interest that a user gives to data objects.

The previous approach is extended in [16], where a context state is represented as a *situation*. Each situation has a *timestamp* that denotes the date and the time of the situation, an entity type *location* that describes the current position and *influences* that describe other aspects affecting the situation. Influences are divided into *personal* and *surrounding*. Personal influences denote human factors of a situation like physical state and surrounding ones describe outer influences like weather condition. In this approach, preferences are mod-

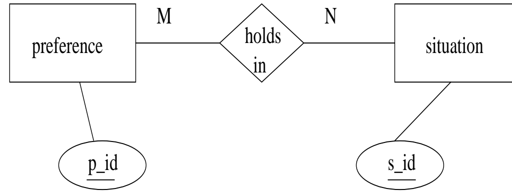

Figure 2.5: The entity relationship schema of Situated Preferences.

eled as in authors' previous work ([14]). There are three types of preferences: $long-term$, $singular$, and $non-singular$. The first one holds generally, while a singular preference holds in exactly one situation. The non-singular preference holds in more than one situation.

Situated preferences are modeled as $N : M$ relationships between situations and preferences (Fig. 2.5). Each situation has a unique *s_id* and each preference has a unique *p_id*. A concrete situated preference can be considered as a tuple (*s_id*, *p_id*) expressing that the preference *p_id* holds in the situation *s_id*.

Chapter 3

The Storage Model

3.1 Storing Basic Preferences

3.2 Storing Context Hierarchies

3.3 Storing the Value Functions

3.4 Storing Aggregate Preferences

In this chapter, we discuss the implementation of our context model in relational DBMS structures. First, we discuss the storage of preferences and then the storage of attribute hierarchies.

3.1 Storing Basic Preferences

There is a straightforward way to store our context and preference information in the database. We organize preferences as data cubes, following the OLAP (On-Line Analytical Processing) paradigm [7]. OLAP is a category of software technology that based on the multidimensional view of data. In this multidimensional data model, *hypercubes*, or simply *cubes*, are used. The information in cubes is stored in a multidimensional array. Thus, a cube is a group of data cells. Each cell is uniquely defined by the corresponding values of the dimensions of the cube. The contents of the cells are named *measures* and represent the measured values of the real world. Measures are functionally dependent, in the relational sense, on the dimensions of the cube.

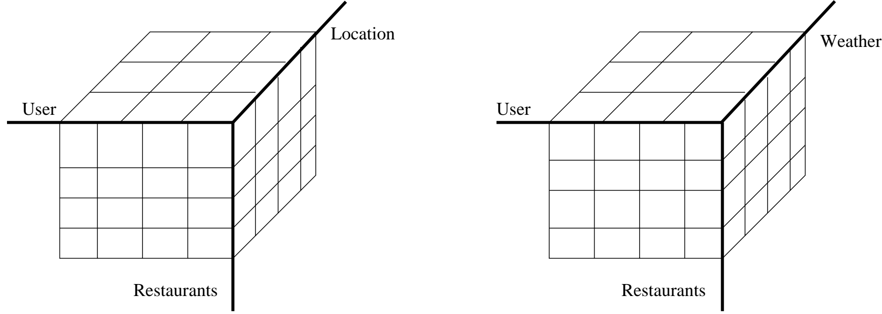

Figure 3.1: Data cubes for each context parameter.

In our model, we store basic user preferences in *cubes*. The number of data cubes is equal with the number of context parameters, i.e., we have one cube for each parameter, as shown in Fig. 3.1. In each cube, there is one dimension for restaurants, one dimension for users and one dimension for the context parameter. In each cell of the cube, we store the degree of interest for a specific preference. So, we can have the knowledge of score for a user, a restaurant and a context parameter. Formally, a *cube* is defined as a finite set of attributes $C = (A_C, A_1, \ldots, A_n, M)$, where A_C is a context parameter, $A_1, \ldots, A_n$ are non-context attributes and M is the interest score. The values of a cube are the values of the corresponding preference rules.

A relational table implements such a cube in a straightforward fashion. The primary key of the table is $A_C, A_1, \ldots, A_n$. If dimension tables representing hierarchies exist (see next), we employ foreign keys for the attributes corresponding to these dimensions.

Our schema is based on the classical *star schema*. In a star schema, data is organized through a centralized fact table, linking several dimension tables. Each dimension table contains information specific to the dimension itself, i.e., information relevant to the dimension. The fact table correlates all dimensions through a set of foreign keys. This is the table that implements the cube. The schema of our approach is depicted in Fig. 3.2. As we can see, there are two fact tables, $Fact_Location$ and $Fact_Weather$. The dimension tables are: $Users$ and *Restaurants*. These are dimension tables for both fact tables.

3.2 Storing Context Hierarchies

An advantage of using cubes to store user preferences is that they provide the capability of using *hierarchies* to introduce different levels of abstractions of

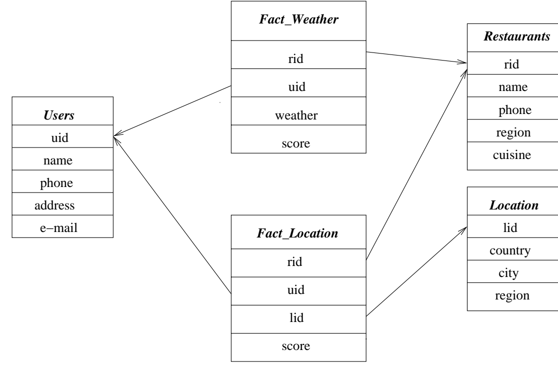

Figure 3.2: The two fact tables of our schema (one for each context parameter) and the dimension tables for $Users$ and $Restaurants$.

the captured context data through the *drill-down* and *roll-up* operators ([17]). By drilling down on the aggregate data the user is getting a more detailed view of the information. Roll-up is the opposite operation: it is the process of viewing data in progressively less detail.

In that way, we can have a hierarchy on a given context dimension. Context dimension hierarchies give to the application the opportunity to use a combination of data between the fact and the dimension tables on one of the context parameters. The typical way to store data in databases is shown in Fig. 3.3 (left). In this modeling, we assign an attribute for each level in the hierarchy. We also assign an artificial key to efficiently implement references to the dimension table. The contents of the table are the values of the $anc_{L_1}^{L_2}$ functions of the hierarchy. The denormalized tables of this kind, participating in a database schema (often called a *star schema*) suffer from the fact that there exists exactly one row for each value of the lowest level of the hierarchy, but no rows explicitly representing values of higher levels of the hierarchy. Therefore, if we want to express preferences at a higher level of the hierarchy, we need to extend this modeling (assume for example that we wish to express the preferences of $Mary$ when she is in $Cyprus$, independently of the specific region, or city of Cyprus she is found at).

To this end, in our model, we use an extension of this approach, as shown in the right of Fig. 3.3. In this kind of dimension tables, we introduce a dedicated tuple for each value at any level of the hierarchy. We populate attributes of lower levels with $NULL$s. To explain the particular level that a value participates at, we also introduce a level indicator attribute. Dimension levels are assigned attribute numbers through a topological sort of the lattice.

G_ID	Region	City	Country
1	Acropolis	Athens	Greece
2	Kefalari	Athens	Greece
3	Perama	Ioannina	Greece
...			

G_ID	Region	City	Country	Level
1	Acropolis	Athens	Greece	1
2	Kefalari	Athens	Greece	1
3	Polichni	Salonica	Greece	1
...				
101	NULL	Athens	Greece	2
102	NULL	Salonica	Greece	2
...				
120	NULL	NULL	Greece	3
121	NULL	NULL	Cyprus	3
...				

Figure 3.3: A typical (left) and an extended dimension table (right).

3.3 Storing the Value Functions

The computation of aggregate preferences refers to the composition of simple basic preferences, in order to compute the aggregate one. The technique used for this involves using weights for each of the parameters. Each aggregate preference involves (a) a set of k context parameters -i.e., cubes and (b) a set of n non-context parameters, common to all context cubes:

$$preference(c_1, \ldots c_k, A_{k+1}, \ldots, A_n) = interest_score$$

The non-context parameters pin the values of the aggregate scores to specific numbers and then, the individual scores for each context parameter are collected from each context table. Recall that the formula for computing an aggregate preference is: $interest_score = w_1 * interest_score_1 + \ldots + w_k * interest_score_k$.

Therefore, the only extra information that needs to be stored concerns the weights employed for the computation of the formula. To this end, we employ a special purpose table $AggScores(w_{C1}, \ldots, w_{Ck}, A_{k+1}, \ldots, A_n)$. The value for each context parameter w_{Ci} is the weight for the respective interest score and the value for each non-context attribute A_j is the specific value uniquely determining the aggregate preference. For instance, in our running example, the table *AggScores* has the attributes *Location_weight*, $Weather_weight$, $User$ and *Restaurant*. A record in this table can be (*0.6, 0.4, Mary, Beau Brummel*). Assume that from $Mary's$ profile, we know that *Beau Brummel* has interest score at the current location 0.8 and at the current weather 0.9, then, the aggregate score is: $0.6 * 0.8 + 0.4 * 0.9 = 0.84$. For simplicity, we just store one weight for each context parameter, making the previous record of the *AggScores* table (*0.6, 0,4, Mary*).

3.4 Storing Aggregate Preferences

Aggregate preferences are not explicitly stored in our system. The main reason is space and time efficiency, since this would require maintaining a context cube for each context state and for each combination of non-context attributes. Assume that the context environment C_X has n context parameters $\{c_1, c_2, \ldots, c_n\}$ and that the cardinality of the domain $dom(c_i)$ of each parameter c_i is (for simplicity) m. This means that there are m^n potential context states, leading to a very large number of context cubes and prohibitively high costs for their maintenance.

Note that some of the m^n context states may not be useful, since they may correspond to combinations of values of context parameters that represent context states that are not valid or have a very small probability of being queried. Furthermore, some context parameters or context states may be more popular for some non-context parameters (e.g., users) than for others, thus making the storage of all states for all non-context parameters unjustifiable. Finally, retrieving specific entries of such cubes is not very efficient, since it would require building and maintaining indexes on various combinations of the context parameters.

For these reasons, we choose to store only previously computed aggregate scores. We also propose using an auxiliary data structure that we call the *context tree* to index them (described in Section 5).

In [18], the Context Relational model (CR model) that uses data cubes is proposed. This model extends the relational model to argue about context. In particular, there is a set of worlds, while each world expresses a combination of values of the context parameters. The number of worlds is equal with the number of all combinations of context parameters' values. A world corresponds to our notion of context state. The authors use cubes to store for each world and for each entity (tuple), the value that each non-context related attribute of the entity has, i.e., there is one dimension for the worlds, one for the entities and one for the attributes (as depicted in Fig. 3.4). In our approach, instead of storing all context states in a cube, we use the context tree to store the states that are previously requested. So, we avoid to store wasteful information, while the number of context states increases exponentially with the number of context parameters.

In order to compare the total storage space of the cubes used in both approaches, we suppose that we store the aggregate preferences in a cube. We consider that we have n context parameters. The cardinality of each domain of each parameter is m. We suppose further, that we have a relation with k attributes, n' of which are context related. This relation has t tuples. In our approach, the cube has one dimension for each context parameter, and one di-

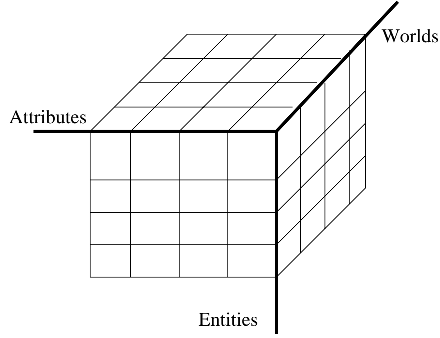

Figure 3.4: A *cube* in the Context Relational model.

mension for the tuples. So, the number of cells is $m^n \times t$. We need also, extra space for storing the t tuples of the relation. The cube in the approach presented in [18] has one dimension for the worlds, one dimension for the tuples and one dimension for the non-context related attributes. Thus, the number of cells is $m^{n'} \times (k - n') \times t$. Comparing the two approaches, note that different information is stored. In our model, we store degrees for data objects in order to rank them according to the user's interest, however in [18] different values are stored for the same objects when the context conditions are different.

Chapter 4

Querying Context

4.1 Querying Simple Preferences
4.2 Querying with Aggregate Scores
4.3 Computing Aggregate Scores
4.4 Traditional OLAP Operators

In this chapter, we classify the query operations that can be posed to our context-aware DBMS, by exploiting the combined information on preferences and context. We further present traditional OLAP operators such as slice, dice, roll-up and drill-down, and how they can be used in our query operations.

4.1 Querying Simple Preferences

Firstly, there are queries executed without a need for the computation of the aggregate score. In this category of queries, users explicitly define that they are not interested in specific context parameters. For example, the following query computes the users' preferences directly.
Query 1 *Look for $Mary's$ most preferable restaurants near Acropolis, independently of the status of weather.*
In SQL, the query is:

- *select* R.name, FL.score
  from Users U, Restaurants R, Fact Location FL, Location L

where U.uid=FL.uid *and* R.rid=FL.rid *and* L.lid=FL.lid *and* U.name='$Mary'$
and L.location='*Acropolis*'
order by FL.score *desc*;

Another similar query would be "Look for the users near Acropolis that prefer restaurant Beau Brummel independently of weather" that can be used to advertise a specific restaurant in the context of "Acropolis".

4.2 Querying with Aggregate Scores

Another useful operation is the computation of aggregate scores from simple ones. For example, the following query needs to compute the aggregate score:
Query 2 *Look for $Mary's$ most preferable restaurants (in the current context).*

The execution of *Query 2* leads to the execution of the following subqueries (we suppose that $CS(current) = \{Acropolis, sunshine\}$):

- *select* R.name, FL.score
  $from$ Users U, Restaurants R, Fact Location FL, Location L
  where U.name='$Mary'$ *and* U.uid=FL.uid *and* R.rid=FL.rid *and* L.lid=FL.lid
  and current location = '*Acropolis*';
  and

- *select* R.name, FW.score
  $from$ Users U, Restaurants R, Fact Weather FW
  where U.name='$Mary'$ *and* U.uid=FW.uid *and* R.rid=FW.rid *and* current weather = '$sunshine'$;

Using the results of subqueries, we calculate the aggregate scores for restaurants using the value function, as described above. In this case, we have the most preferable Mary's restaurants in decreasing order.

4.3 Computing Aggregate Scores

The technique used for processing queries involving aggregate scores (e.g., *Query 2* above) is the following.

1. First, we select specific values for $Users$ and for the context parameter. For instance, for the first cube a selection could be on a value of $location$, e.g., *Acropolis* and for a value of $user$, e.g., $Mary$.

2. Second, having pinned all dimension attributes to a specific value, we have all the preference interest scores available. In fact, the individual scores

for each context parameter are collected from each context table (although this practically involves a relational join on all non-context parameters, it is quite more easy to simply collect the values from the respective cubes from a set of point queries over them). So, we can compute the aggregate score of a preference by using a value function (as described in the previous sections).

3. In the context of an OLAP session, the aggregate scores just computed for a user can be stored in a new transient cube. As with cubes concerning basic preferences, a cube concerning aggregate preferences has one attribute for each context and non-context parameter and an extra attribute for the interest score. Then, the user can reuse the result of a query, by just using the last cube, without executing all the above steps. In Section 5, we describe a space-efficient structure for storing such results.

4.4 Traditional OLAP Operators

OLAP provides a principled way of querying information. The traditional techniques for relational querying are enriched with special purpose query operators, such as roll-up and drill-down [19, 7].

Slice-n-Dice. The *dice* operator on a data set corresponds to a selection (in the relational sense) of values on each dimension. A *slice* is a selection on one of the N dimensions of the cube. A dice operator can be implemented as a sequence of slices. Simple preference queries can be computed using slice operators. For instance, *Query 1* can be implemented using slice operations on $User$ and *Location*.

Roll-up. The *roll-up* operation provides an aggregation on one dimension. More specific, roll-up corresponds to the aggregation of data from a lower to a higher level of granularity within a dimension's hierarchy. Assume that the user has executed *Query 1* over the database and receives an unsatisfactory small number of answers. Then, she can decide that is worth broadening the scope of the search and investigate the broader Athens area for interesting restaurants. In this case, a *roll-up* operation on *location* can generate a cube that uses *cities* instead of $regions$. The following query express this roll-up operation in SQL:

- $select$ R.name, FL.score
  $from$ Users U, Restaurants R, Fact Location FL, Location L

where U.uid=FL.uid *and* R.rid=FL.rid *and* L.lid=FL.lid *and* U.name = '*Mary*' *and* L.city = '*Athens*'
order by FL.score *desc*;

Drill-down. Similarly, *drill-down* is the reverse of roll-up and allows the de-aggregation of information moving from higher to lower levels of granularity. So, when we have the result of a query which includes *restaurants* that are located in *Athens*, we can take a result that includes *restaurants* located at *Acropolis*, using the *drill-down* operator.

Chapter 5

Caching Context States

5.1 The Context Tree
5.2 Querying the Context Tree
5.3 Querying with Approximate Results
5.4 Querying Using Hierarchies
5.5 Additional Issues
5.6 Bloom-Based Index for the Context Tree

In this chapter, we discuss issues regarding improving the performance of our system. In particular, we discuss how we can store (cache) results of previous queries executed at a specific context, so that these results can be re-used by subsequent queries. First, we describe a hierarchical data structure, called *context tree*, that is used to index these results. Then, we show how search in this data structure can be improved using an additional hash-based index to test for membership in the context tree.

5.1 The Context Tree

Assume that the context environment C_X has n context parameters $\{c_1, c_2, \ldots, c_n\}$. An alternative way to store aggregate preferences uses a *context tree*, as shown in Fig. 5.1. The context tree is used to store aggregate preferences that were computed as results of previous queries, so that these results can be

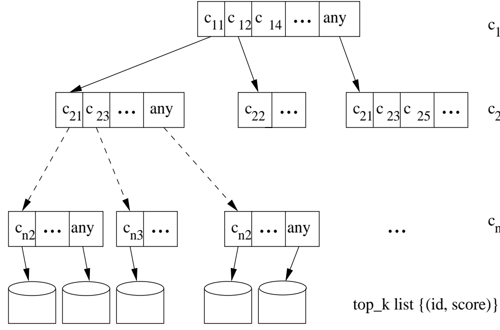

Figure 5.1: A context tree.

re-used by subsequent ones. There is one context tree per user or per system-defined profile (i.e., per group of users with similar interests, see Section 2.5). The maximum height of the context tree is equal to the number of context parameters plus one. Each context parameter is mapped onto one of the levels of the tree and there is one additional level for the leaves. For simplicity, assume that context parameter c_i is mapped to level i. A path in the context tree denotes a *context state*, i.e., an assignment of values to context parameters.

At the leaf nodes, we store a list of ids, e.g., restaurant ids, along with their aggregate scores for the associated context state, that is, for the path from the root leading to them. Instead of storing aggregate values for all non-context parameters, to be storage-efficient, we just store the $top - k$ ids (keys), that is the ids of the items having the k-highest aggregate scores for the path leading to them. The motivation is that this allows us to provide users with a fast answer with the data items that best match their query. Only if more than k-results are needed, additional computation will be initiated. The list of ids is sorted in decreasing order according to their scores.

The context tree is constructed incrementally each time a context-aware query is computed. Each non-leaf node at level k contains cells of the form $[key, pointer]$, where *key* is equal to $c_{kj} \in dom(c_k)$ for a value of the context parameter c_k that appeared in some previously computed context query. The pointer of each cell 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 query with c_{kj}. In addition, *key* may take the special value *any*, which corresponds to the lack of the specification of the associated context parameter in the query. For example, assume two context parameters, *location* and *weather* and that *weather* is assigned to level m of the tree and *location* to the level just below it, level $m+1$. Then, take for instance a query, where the user specifies $weather = cloudy$, but gives no value

query 1 / cloudy / Plaka
query 2 / cloudy / Acropolis
query 3 / sunshine / Plaka
query 4 / cloudy / *

Figure 5.2: A set of aggregate preferences.

for location. Then, there will be a cell $[cloudy, pointer]$ at level m pointing to a node at level $m+1$ containing a cell $[any, pointer]$. Initially, the context tree is empty, that is the root node contains a single cell of the form $[any, null]$.

The way that the context parameters are assigned to the levels of the context tree affects its size. As a simple heuristic, context parameters are assigned to levels based on the cardinality of their domains: the bigger the number of distinct values a context parameter takes, the higher it appears in the context tree. We explain further the mapping of context parameters to levels in Section 5.5.2.

In Fig. 5.2, we present a set of context preferences expressed in four previously submitted queries. Assume that we have two context parameters, *weather* and *location* and that *weather* is assigned to the first level of the tree and *location* to the next one. Leaf nodes store the ids of the $top-k$ restaurants, that is the restaurants with the $top-k$ highest aggregate scores. Related work has been done in the area of computing the $top-k$ objects for a query's result in [20, 21, 22]. This work is presented in chapter 7. For the above preference queries we construct the context tree of Fig. 5.3.

In this tree, for the first preference (*cloudy*/*Plaka*) we construct the first path of the tree (the leftmost one). The next preference (*cloudy*/*Acropolis*) has the same value for the context parameter *weather* with the first one, so we do not add any new cell to the root node. Next, we add a cell for *Acropolis* to the node that the root points to. The third preference (*sunshine*/*Plaka*) has value *sunshine* for *weather* and this leads to the creation of a new cell in the root node. As before, the last preference (/*cloudy*/$*$) has a *cloudy* value and so, the remaining path for that preference has as a predecessor the *cloudy* cell of the root. If we follow the *any* cell of the corresponding node of the second level, the $*$ operator of this preference is satisfied.

The aggregate preferences in Fig. 5.2 can be derived from simple users' queries that have presented in the previous chapter. For instance, the query "*Look for Mary's most preferable restaurants in the current context*, while CS(current) = $\{cloudy, Plaka\}$", corresponds to the first preference of Fig. 5.2. Respectively, the preference (*cloudy/**) can be expressed by the query "*Look for Mary's most preferable restaurants, when the weather is cloudy, independently*

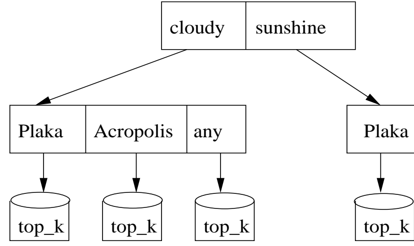

Figure 5.3: A context tree for a specific profile.

of her location". This query in SQL is:

- *select* R.name, FW.score
  from Users U, Restaurants R, Fact Weather FW
  where U.uid=FW.uid *and* R.rid=FW.rid *and* U.name='$Mary'$
  and weather='$cloudy'$
  order by FW.score *desc*;

When a query is issued, we first check whether there exists a context state that matches it in the context tree. If so, we retrieve the $top - k$ results from the associated leaf node. Otherwise, we compute the answer and insert the new context state in the tree. There is a number of interesting variations. For instance, if the query includes the * operator for a context parameter, we can combine the $top{-}k$ results that are associated with the paths that have any value for this parameter and the same values for all the others, to produce the new list of $top - k$ results. Furthermore, instead of storing the results of all queries, we may just store the results of the most frequently requested ones. This can be easily implemented by associating a counter with each path and replacing (deleting) from the tree the path that is less frequently used. Other issues refer to queries with approximate results or to queries that their context parameters participate in different levels of hierarchy. All these issues are illustrated further in the following sections.

In summary, a context tree for n context parameters satisfies the following properties:

- It is a directed acyclic graph with a single root node.
- There are at most $n{+}1$ levels, each one of the first n of them corresponding to a context parameter and the last one to the level of the leaf nodes.
- Each non-leaf node at level k maintains cells of the form $[key, pointer]$ where $key \in dom(c_k)$ for some value of c_k that appeared in a query or key

= *any*. No two cells within the same node contain the same key value. The pointer points to a node at level $k + 1$ having cells with key values which appeared in the same query with the key.

- Each leaf node stores a set of sorted pointers to data.

The context tree resembles the Dwarf data structure [23] used to compute and store data cubes. Whereas Dwarf is build by scanning the fact table and includes all existing combinations of values for all cube dimensions, the context tree is incrementally computed, each time a preference query is evaluated and includes only paths (context states) previously queried. Furthermore, Dwarf leaf nodes contain aggregate values, whereas the context tree leaf nodes contain ordered sets.

5.2 Querying the Context Tree

The context tree provides an efficient way to retrieve the *top-k* results that are relevant to a preference query. When a query is posed to the system, we first check if there exists a context state that matches it in the context tree. If so, we retrieve the *top-k* results from the associated leaf node. Otherwise, we compute the answer and insert the new context state, i.e., the new path and the associated *top-k* results, in the tree.

However, in some cases, even if a path corresponding to a query does not exist in the context tree, there is no need to compute the answer from scratch for all kinds of queries. Instead, it may be possible to derive the *top-k* results from a combination of other relative results. In our approach, we use this method when a '*' operator is contained in the query. More specifically, when a query includes the '*' value for a context parameter, we merge the *top-k* results of the paths that have any value for this parameter and the same values for all the others, in order to take the *top-k* results of the new query.

For example, for the context tree of Fig. 5.3, when we have the query */Plaka*, we execute a merge sort algorithm to merge the results of the first and the last path. Next, we insert the new context state in the tree. From this new path, we can retrieve the combined *top-k* results.

As referred above, a query is a simple traversal on the context tree from the root to a leaf. At level i, we search for the cell having as key the ith value in the query and descend to the next level. If the ith value is *any* (respectively the '*' operator), we follow the pointer of the *any* cell. For a context tree with n context parameters $(c_1, c_2, \ldots, c_n)$, if each parameter has $|dom(c_i)|$ values in its domain, the maximum number of cells that are required to be visited for

a query is $|dom(c_1)| + |dom(c_2)| + \ldots + |dom(c_n)|$. Each query is fast, simply, because it involves exactly such node visits as the height of the tree minus one, i.e., as the number of the context parameters. Algorithm 1 below presents the search process in the context tree.

Algorithm 1 Search_Path_Algorithm

Input: *node*, *query*[]
Output: $top - k$ *results*

```
if query[ctr] != -1 then
  /* ctr refers the level of the query */
  while not all the existing cells are examined do
    if the key of a cell is equal with query[ctr] then
      ctr++;
      if this is the last level of the query then
        return the top - k list
      end if
      call Search_Path_Algorithm (node → child, query)
      return
    end if
  end while
end if
```

5.3 Querying with Approximate Results

We further extend our approach by checking when the $top - k$ results can be derived from a combination of relative results for a query with a '*' operator. So, for a specific context parameter, if its value is a '*', we check if the number of the existing values of the same parameter in the same node in the context tree is larger than a threshold value. Only in this case, i.e., in the case that there are more than $x\%$ values of the corresponding parameter in the tree, we merge the relative results. Otherwise, we compute the answer. The threshold $x\%$ is called *approximation coverage threshold*. The value for the threshold parameter may be either system defined or given as input by the user.

For instance, for the previous query (**/Plaka*), the corresponding node in the context tree (Fig. 5.3) has two of the three values of the weather's domain. We remind that this domain has the values: *rainy*, *cloudy*, and *sunshine*. So, if the approximation coverage threshold has a value that is less or equal to 66.67% the result comes as before. Otherwise, we compute the $top - k$ results from scratch.

An alternative method supports queries that give as answers approximate results. So, instead of descending to the next level of the tree only if a cell has the same value with the corresponding query's value, we continue to traverse the tree if a cell has a relative to the query's value and not the same one. We introduce a *neighborhood approximation threshold* to express when two values are relative. The value for this threshold is different for each context parameter and depends on its domain. As before, the threshold may be either determined by the user or is system defined. In any case, it is necessary to store some extra information that concerns the values that are approximate values with others.

For example, the query *rainy/Plaka* can give us the results of the first *top-k* list, (Fig. 5.3), if a user considers that the values *rainy* and *cloudy* are similar. As depicted in Fig. 5.3, these results are associated with the query *cloudy/Plaka*.

Furthermore, we are interested to know how the transition from a context state to another affects the aggregate scores of the tuples. In particular, in order to show how much similar the *top-k* results are, when two approximate queries are posed in the system, we first prove the intuitive property that even "small" changes in context values may lead to different rating. Then, we prove that "small" changes in context values between a context state s and a context state s' leads to "small" changes in the rating of the queries' results.

Suppose for example, two context parameters as relevant: *weather* and *location*. Assume that weather has weight $w_1 = 0.6$ and location has weight $w_2 = 0.4$. Also, suppose results that include two tuples: the restaurant $Ithaka$ (t_1) and the restaurant *Golden Lake* (t_2). We assign degrees to preferences as follows. Tuple t_1 has degree of interest 0.9 when the weather is *sunshine*, 0.7 for *cloudy* weather and 0.6 for *rainy* weather. Also, t_1 has degree 0.9 when the user is in *Ioannina* and 0.7 when the user is at *Anatoli*. Tuple t_2 has degrees 0.6, 0.8, 0.9 when we have *sunshine*, *cloudy* and *rainy* weather, respectively. Finally, the degrees of interest are 0.6 and 0.8, when the user is in *Ioannina* or at *Anatoli*.

Note that we suppose that the values *cloudy* and *rainy* for the context parameter weather are similar and so, their scores have nearby values. In that way, for the query cloudy/Ioannina, t_1 has aggregate score $w_1 * d_1 + w_2 * d_2 = 0.6 * 0.7 + 0.4 * 0.7 = 0.78$ and t_2 has score $0.6 * 0.8 + 0.4 * 0.6 = 0.72$. For the similar query rainy/Ioannina, t_1 has score 0.72 and t_2 0.78. Consequently, as referred above, small changes in context values leads to a different rating of the queries' results.

Next, we prove that "small" changes in context values lead to "small" changes in the rating of the queries' results.

More specifically, let t_1 and t_2 be two tuples and s and s' be two context states. Assume that in s, t_1 and t_2 have aggregate scores d_1 and d_2 respectively and in s', d'_1 and d'_2 respectively. When the two context states are similar, we

would like the following to hold:

$$|d_1 - d_2| \leq \varepsilon \Rightarrow \tag{5.1}$$

$$|d'_1 - d'_2| \leq \delta \tag{5.2}$$

where ε and δ are "small" positive constants.

Assume further, that if two context states have similar values, then the scores of the tuples are similar that is, for every t and for a specific context parameter,

$$|d_{it} - d'_{it}| \leq \varepsilon' \tag{5.3}$$

for a small constant ε', with $0 \leq \varepsilon' \leq 1$.

With the following property, we prove that when only one of the degrees of the context parameters is changed, $\delta = \varepsilon + 2 * w_1 * \varepsilon'$, where ε' is the difference between the degrees of interest of the same context parameter of two similar context states, and w_1 is the weight of the context parameter that its value is changed.

Property 1 *Let t_1, t_2 be two tuples that have aggregate scores d_1, d_2 in a context state s and d'_1, d'_2 in a context state s' respectively. When s, s' are similar, i.e., $|d_{it} - d'_{it}| \leq \varepsilon'$, and only one of the degrees of the context parameters is changed, if $|d_1 - d_2| \leq \varepsilon$ then, $|d'_1 - d'_2| \leq \varepsilon + 2 * w_1 * \varepsilon'$.*

Proof Since only one of the degrees of the context parameters is changed, we have:

$$d_1 = w_1 * d_{1t_1} + \Sigma_{i=2}^{n} w_i * d_{it_1} \tag{5.4}$$

$$d_2 = w_1 * d_{1t_2} + \Sigma_{i=2}^{n} w_i * d_{it_2} \tag{5.5}$$

and

$$d'_1 = w_1 * d'_{1t_1} + \Sigma_{i=2}^{n} w_i * d_{it_1} \tag{5.6}$$

$$d'_2 = w_1 * d'_{1t_2} + \Sigma_{i=2}^{n} w_i * d_{it_2} \tag{5.7}$$

From (5.3), for the above two tuples we have:

$$|d_{1t_1} - d'_{1t_1}| \leq \varepsilon' \tag{5.8}$$

$$|d_{1t_2} - d'_{1t_2}| \leq \varepsilon' \quad (5.9)$$

From (5.4), (5.5) and (5.6), (5.7), Equations (5.1), (5.2) can be written as:

$$|w_1 * d_{1t_1} + \Sigma_{i=2}^{n} w_i * d_{it_1} - w_1 * d_{1t_2} - \Sigma_{i=2}^{n} w_i * d_{it_2}| \leq \varepsilon \quad (5.10)$$

$$|w_1 * d'_{1t_1} + \Sigma_{i=2}^{n} w_i * d_{it_1} - w_1 * d'_{1t_2} - \Sigma_{i=2}^{n} w_i * d_{it_2}| \leq \delta \quad (5.11)$$

and so, we would like to prove that if (5.10) holds, then Equation (5.11) holds, with the assumptions of (5.8) and (5.9).

From (5.10), we have: $-\varepsilon \leq w_1 * d_{1t_1} + \Sigma_{i=2}^{n} w_i * d_{it_1} - w_1 * d_{1t_2} - \Sigma_{i=2}^{n} w_i * d_{it_2} \leq \varepsilon \Rightarrow$

$$a \leq w_1 * d'_{1t_1} + \Sigma_{i=2}^{n} w_i * d_{it_1} - w_1 * d'_{1t_2} - \Sigma_{i=2}^{n} w_i * d_{it_2} \leq b \quad (5.12)$$

with $a = -\varepsilon - w_1 * d_{1t_1} + w_1 * d'_{1t_1} + w_1 * d_{1t_2} - w_1 * d'_{1t_2}$ and $b = \varepsilon - w_1 * d_{1t_1} + w_1 * d'_{1t_1} + w_1 * d_{1t_2} - w_1 * d'_{1t_2}$.

From (5.8), and (5.9) we can take that: $d_{1t_2} - d'_{1t_2} \geq -\varepsilon'$ and $d'_{1t_1} - d_{1t_1} \geq -\varepsilon'$, and so, $a = -\varepsilon + w_1 * (d'_{1t_1} - d_{1t_1}) + w_1 * (d_{1t_2} - d'_{1t_2}) \geq -\varepsilon - w_1 * \varepsilon' - w_1 * \varepsilon' = -\varepsilon - 2 * w_1 * \varepsilon'$, i.e., $a \geq -\varepsilon - 2 * w_1 * \varepsilon'$.

Furthermore, from (5.8), and (5.9) we can take that: $d_{1t_2} - d'_{1t_2} \leq \varepsilon'$ and $d'_{1t_1} - d_{1t_1} \leq \varepsilon'$, and so, $b = \varepsilon + w_1 * (d'_{1t_1} - d_{1t_1}) + w_1 * (d_{1t_2} - d'_{1t_2}) \leq \varepsilon + w_1 * \varepsilon' + w_1 * \varepsilon' = \varepsilon + 2 * w_1 * \varepsilon'$, i.e., $b \leq \varepsilon + 2 * w_1 * \varepsilon'$.

From the above, (5.12) can be written: $-\varepsilon - 2 * w_1 * \varepsilon' \leq w_1 * d'_{1t_1} + \Sigma_{i=2}^{n} w_i * d_{it_1} - w_1 * d'_{1t_2} - \Sigma_{i=2}^{n} w_i * d_{it_2} \leq \varepsilon + 2 * w_1 * \varepsilon' \Rightarrow$

$$|w_1 * d'_{1t_1} + \Sigma_{i=2}^{n} w_i * d_{it_1} - w_1 * d'_{1t_2} - \Sigma_{i=2}^{n} w_i * d_{it_2}| \leq \varepsilon + 2 * w_1 * \varepsilon' \quad (5.13)$$

and thus, the equation (5.11) holds, with $\delta = \varepsilon + 2 * w_1 * \varepsilon'$.

We generalize Property 1 for the case where more than one of the degrees of the context parameters are changed. Thus:

Property 2 *Let t_1, t_2 be two tuples that have aggregate scores d_1, d_2 in a context state s and d'_1, d'_2 in a context state s' respectively. When s, s' are similar, i.e., $|d_{it} - d'_{it}| \leq \varepsilon'$, if $|d_1 - d_2| \leq \varepsilon$ then, $|d'_1 - d'_2| \leq \varepsilon + 2 * (w_1 + w_2 + \ldots + w_n) * \varepsilon'$.*

The proof of Property 2 proceeds similar to the proof of Property 1.

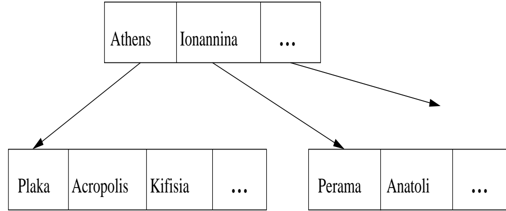

Figure 5.4: The hierarchy tree of *location*.

5.4 Querying Using Hierarchies

A extension of our model supports hierarchies in posing and answering context-aware queries. With this extension, a query may contain a context parameter that participates in a non leaf level of the hierarchy. When a query is issued and the value of a context parameter is given in an upper level of the hierarchy, we combine the *top-k* results that are associated with paths that contain the corresponding values of the given value in the query.

The only extra information that needs to be stored concerns the hierarchy of the context parameter. We store this hierarchy in a tree. The height of the hierarchy tree is equal to the number of the hierarchy levels minus 1. Each node of the tree contains cells of the form [*key, pointer*], where key is equal to a value of the corresponding level of the hierarchy. The pointer of each cell points to the node at the next lower level.

In the following example, we show how this variation of the context tree is used. Figure 5.4 represents the hierarchy tree for the context parameter *location*. In the root of the tree, we store information about cities in *Greece*. Each cell of the root points to specific regions of a city. So, for the context tree of Fig. 5.3, when we have the query *cloudy/Athens*, we retrieve the regions of *Athens* from the hierarchy tree and then we merge the *top-k* results that are associated with the context states that have for weather the value *cloudy* and for location values retrieved from the hierarchy tree. In that way, the system returns a list of *top-k* results.

5.5 Additional Issues

5.5.1 Replacement Policies

The context tree is an alternative structure to the cube. We use it to store aggregate preferences that were computed as results of previous queries, so that these results can be re-used by subsequent ones. The main reason is space

and time efficiency. This tree is constructed incrementally. In that way, each time a preference query is evaluated, the path that corresponds to the context state of the query is added to the tree. In order to keep the tree in memory, it is desirable to take up limited space. Thus, instead of storing the results of all context-aware queries, we may just store the results of the most frequently requested ones. This variation of the context tree improves query performance, because in this case less time is required for traversing the context tree.

In general, the context tree is used to cache context states that are related with previous submitted queries. This tree can store a specific number of context states, and so, it may be needed to delete some of them. We consider two replacement policies for choosing which paths to delete from the context tree. The first one uses the LRU (*Least Recently Used*) algorithm: we replace the path that was used least recently. To implement this policy, we store the time of the last access of a path at each path of the tree and we replace - delete the path with the oldest (smallest) such value. The second replacement policy uses the LFU (*Least Frequently Used*) algorithm: we replace the path that is least frequently used. This can be easily implemented by associating a counter with each path. The counter maintains the number of times that the corresponding path is accessed. So, we delete from the tree the path having the counter with the smallest value.

5.5.2 Mapping Context Parameters to Levels

The choice of the ordering of the context parameters has an effect on the total size of the context tree. In particular, context parameters with higher cardinalities in their domains are more beneficial if they are placed on the higher levels of the context tree. Using this mapping, we eliminate the total storage space of the context tree, because the total number of cells is smaller. For instance, if the domain of the first level of the tree, i.e., the root of the tree, has n_0 values, the second level has n_1 values, and the last one n_k, the total number of cells is $n_0 * (1 + n_1 * (1 + \ldots (1 + n_k)))$. Thus, when $n_0 \leq n_1 \leq \ldots \leq n_k$ the above number is as small as possible.

This way of assigning parameters to levels is a simple heuristic method. Unfortunately, the best mapping depends on the query workload. For example, if we have many queries with a '*' operator for a specific context parameter, it is efficient to assign this parameter to the highest level of the tree. Consequently, the semantics of each parameter in combination with the query workload of the system constitutes a basic factor for assigning context parameters to levels.

The ordering used will either be the one given by the user (if one has been specified), or will be automatically chosen by the system after determining the

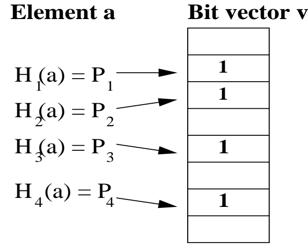

Figure 5.5: A Bloom filter with 4 hash functions.

cardinalities of the parameters domains and potentially the query workload.

5.6 Bloom-Based Index for the Context Tree

In order to improve the query performance of the context tree, we propose using Bloom filters [24]. A Bloom filter is a main-memory data structure that supports very efficient membership queries. When a new query for a context state is submitted by the user, instead of searching the context tree for a matching context state, the Bloom-based data structure is conducted first. Given a context state, the Bloom-based data structure provides a quick answer on whether this state exists in the tree. If the context state does not exist, then retrieving the entire context tree is avoided.

5.6.1 Bloom Filters Preliminaries

A Bloom filter is a space-efficient probabilistic data structure that is used to test whether or not an element is a member of a set. This method is used for representing a set $A = a_1, a_2, \ldots, a_l$ of l elements (also called keys) to support membership queries (*is element X in set Y?*). The idea is to allocate a vector v of m bits (Fig. 5.5), initially all set to 0, and then choose f independent hash functions, $h_1, h_2, \ldots, h_f$, each with range 1 to m. For each element A, the bits at positions $h_1(a), h_2(a), \ldots, h_f(a)$ in v are set to 1. A particular bit might be set to 1 multiple times, but only the first change has an effect. Given a query for b we check the bits at positions $h_1(b), h_2(b), \ldots, h_f(b)$. If any of them is 0, then certainly b is not in the set A. Otherwise we conjecture that b is in the set although there is a certain probability that we are wrong. This is called a *false positive* (or a false drop) and it is the payoff for Bloom filters' compactness. The parameters k and f should be chosen such that the probability of a false positive (and hence a false hit) is acceptable. Although false positives are possible, false negatives are not.

BBF_1	1	0	0	0	1	0	1	0	0	0	0	0	→ cloudy OR sunshine OR any
BBF_2	0	1	1	1	1	0	0	0	1	0	0	1	→ Plaka OR Acropolis OR any

Figure 5.6: The BBF for the context tree of Fig 5.3.

The probability of a false positive for an element not in the set, or the false positive rate, can be calculated in a straightforward fashion, given our assumption that hash functions are perfectly random. After all the elements of A are hashed into the Bloom filter, the probability that a specific bit is still 0 is $(1-1/m)^{f\,l} \simeq e^{-f\,l/m}$, where m is the size of the Bloom filter, f is the number of hash functions and l is the number of elements that we index in the filter.

In their original form, Bloom filters provided support only for simple keyword queries and not for path queries such as those representing context states. To this end, in our previous work we have introduced multi-level Bloom filters, namely *Breadth* and *Depth Bloom filters* [25, 26].

5.6.2 Multi-level Bloom Filters

Let a context tree T for n context parameters and let the level of the root be level 1. There are two ways to hash the tree, corresponding to its breadth and depth first traversal.

The *Breadth Bloom Filter* (BBF) for a context tree T with n context parameters is a set of n Bloom filters $\{BBF_1, BBF_2, \ldots, BBF_i\}$, where each Bloom filter, BBF_i, corresponds to an internal (i.e., non leaf) level i of the context tree, that is, there is one filter for each context parameter c_i. In each BBF_i, we insert all *keys* that appear in cells in nodes at level i of the context tree. For example, the BBF for the context tree of Fig. 5.3 is a set of two Bloom filters (Fig. 5.6).

Depth Bloom filters provide an alternative way to summarize context trees. We use different Bloom filters to hash paths of different lengths. The *Depth Bloom Filter* (DBF) for a context tree T for n context parameters is a set of Bloom filters $\{DBF_0, DBF_1, DBF_2, \ldots, DBF_{i-1}\}$, $i \leq n$. There is one Bloom filter, denoted DBF_i, for each path of the tree with length i, that is, having $(i+1)$ nodes, where we insert all paths of length i. Note that a path with length $i < n$ corresponds to a context state in which there are values specified only for the first i parameters. Note that we insert paths as a whole, we do not hash each element of the path separately; instead, we hash their concatenation. The DBF for the context tree in Fig. 5.3 is a set of two Bloom filters (Fig. 5.7).

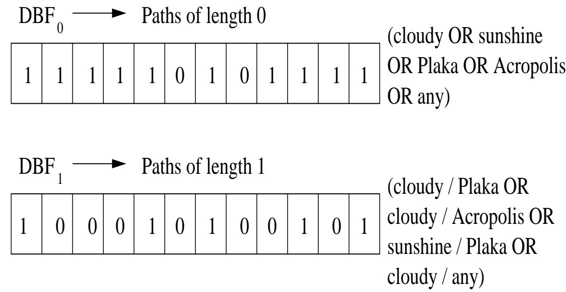

Figure 5.7: The DBF for the context tree of Fig 5.3.

Let a path $a_1/a_2/\ldots/a_p$ corresponding to a context state with p context parameters. In the case of a BBF, to check whether a path $a_1/a_2/\ldots/a_p$ corresponding to a context state exists, each level i from 1 to p of the filter is checked for the corresponding a_i. The test is positive, if we have a hit for all elements in the path. In the case of DBF, we first check whether all elements in the path expression appear in DBF_0. Then, for a query of length p, every sub-path of the query with length 2 to p is checked at the corresponding level. If we have a match for all sub-paths, then we conclude that the path may exist in the context tree, else we have a miss.

When comparing the two filters, DBF works better (has a smaller false positive ratio) than BBF [25]. The reason is that when using $BBFs$, a new kind of false positive appears. Consider the tree of Fig. 5.3 and the query: *sunshine/Acropolis*. We have a match for *sunshine* at BBF_1 and for *Acropolis* at BBF_2; thus we falsely deduce that the path exists. However, DBFs is less space efficient, since the number of paths is very large. This the reason, that we may keep Bloom filters for paths with length less than $n-1$.

Chapter 6

Implementation and Evaluation

6.1 Prototype Implementation

6.2 Performance Evaluation of the Context Tree

6.1 Prototype Implementation

Figure 6.1 depicts the overall system architecture of a preference database system. The Context-Aware Preference Database Management System (DBMS) stores both database relations and preferences that relate the context-dependent attributes of the relations with the context parameters. To process context-dependent queries, preferences are taken into account to present the results based on their preference score at the specified context state. We assume that the values of the current context state are provided as input to our system.

To demonstrate the feasibility of our approach, we have developed a prototype application based on our reference example. The application is called *Preference Restaurant Guide* and maintains information about restaurants and users. Its schema is the one depicted in Fig. 3.2. We consider two context parameters as relevant: *location* and *weather*. The prototype application is build on top of Oracle 8i, using Borland JBuilder 7. The prototype implements all modules of our approach except of the context tree.

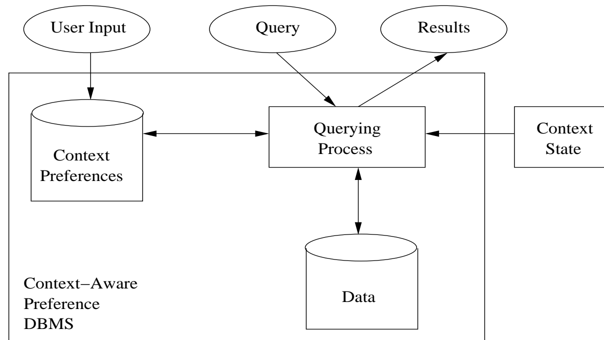

Figure 6.1: Overall system architecture of a Context-Dependent Preference Database.

When a user joins the system, she registers her attributes and then, she selects which context parameters she considers as relevant. Users express their preferences about restaurants by providing a numerical score between 0 and 1. The degree of interest that a user expresses for a restaurant depends on the values of the context parameters, she considers as relevant. If more than one context parameter is defined as relevant, i.e., both *location* and $weather$, weights are specified to express how each parameter affects the computation of the aggregate score.

Besides user registration, the other part of the application includes query processing. Query processing runs in two modes: context-aware and non context-aware. In the non-context aware node, preferences are ignored. In the context-aware mode, the user specifies the values of the context-parameters she is interested in, and the results are sorted according to the user's preference in the specified context state. There is a default state that corresponds to the current context state. In addition, a user may use an OLAP operator to execute a *roll-up* or a *drill-down* to the results of a query. For example, suppose a result that contains restaurants located in the region of *Acropolis*. A single $roll-up$ provides restaurants in the city of *Athens*. In following, we present a sequence of instances of the application *Preference Restaurant Guide*.

Suppose that $Mary$ is at $Kifisia$ and the weather is *sunshine*, i.e., the current context state is CS(current) = $\{Kifisia, sunshine\}$, when she would like to know the best two restaurants, according to her preferences that are located at $Kifisia$. The above query is expressed as shown in Fig. 6.2 and the result is depicted in Fig. 6.3. If $Mary$ chooses to use an OLAP operator to execute a $roll-up$ to the results of the query, the application returns the restaurants that are located in the city of *Athens*. The results of this operation

Figure 6.2: Query Example.

show restaurants

R_NAME	R_ID	U_ID	AGG_SCORE
PIAZZAMELA	23	1	0.90
BEAU BRUMMEL	3	1	0.86

Figure 6.3: Result Example.

are shown Fig. 6.4.

6.2 Performance Evaluation of the Context Tree

In this section, we evaluate the performance of the context tree on answering queries. We run a set of experiments with different values on the input parameters. The input parameters of our experiments are summarized in Table 6.1.

We divide the input parameters into three categories: *context parameters*, *query workload parameters*, and *query approximation parameters*. In particular,

show restaurants

R_NAME	R_ID	U_ID	AGG_SCORE_OLAP
PIAZZAMELA	23	1	0.90
BEAU BRUMMEL	3	1	0.86
VARDIS	6	1	0.80
MILOS ESTIATORIO	12	1	0.78
BIG DEALS	16	1	0.76
INTERNI	19	1	0.72
HYTRA	10	1	0.68
THE RESTAURANT	13	1	0.60
EDWDH	18	1	0.54

Figure 6.4: Result Example.

we use three context parameters as relevant and thus, the context tree has three levels (plus one for the $top-k$ lists). There are two different number of sizes for the *cardinalities* of the domains of the context parameters: the *small* domain with 5 values and the *large* one with 50 values. Furthermore, we use a *hierarchy tree* with three levels. This tree is associated with one of the context parameters.

The *approximation coverage threshold* refers to the percentage of values in a node that need to be present for a context parameter, in order to compute the $top - k$ list combining other computed lists when there is a '*' operator at the corresponding level in a new query. In a similar way, the *neighborhood approximation threshold* refers to the approximate values that are relative to the values of the queries, while the parameter ε' expresses the difference between the degrees of interest of the same context parameter of two similar context states. Furthermore, we use various values for *weights*. In particular, there are two cases: at the first one all the weights have the value 0.33, and at the second one the weights w_1, w_2, and w_3 shared the values 0.5, 0.3, and 0.2.

Finally, we performed our experiments with various numbers of *stored queries* from 50 to 200 with a step of 50 queries, while the number of tuples is 1000. 10% of the values are '*'. The other 90% are either *uniformly random* values from the domain of the relative context parameter, or follow the *zipf* data distribution. Using this distribution, we can select the values that are more frequent in queries, and changing the value of parameter a, we can specify how frequent these values are. So, for a domain with 5 values (1 to 5), when the ranking determines that 1 is more frequent than 2, 2 more frequent than 3, and so on, the frequencies of the values, when we select 100 of them, are represented in Fig.6.5. If $a = 1.5$, the first values of the ranking are more frequent, than when

Table 6.1: Input Parameters

Context Parameters	**Default Value**	**Range**
Number of Context Parameters	3	
Cardinality of the Context Parameters' Domains		
Small	5	
Large	50	
Levels of hierarchy	3	
Query Workload		
Number of Tuples	1000	
Number of Stored Queries		50-200
Percentage of '*' values	10%	
Data Distributions	$uniform$	
	$zipf$ - a=1.0	
	$zipf$ - a=1.5	
Query Approximation		
Approximation Coverage Threshold	$\geq 40\%$, $\geq 60\%$, $\geq 80\%$	
Neighborhood Approximation Threshold		
For a *Small* Domain		
If v_i odd	$v_i + 1$	
Else	$v_i - 1$	
For a *Large* Domain		
If v_i' mod 5 $! = 0$		$[\lfloor v_i'/5 \rfloor * 5 + 1, \lceil v_i'/5 \rceil * 5]$
Else		$[(v_i'/5 - 1) * 5 + 1, v_i']$
Parameter ε'	0.04	
	0.08	
	0.12	
Different Weights	0.5, 0.3, 0.2	
	0.33	

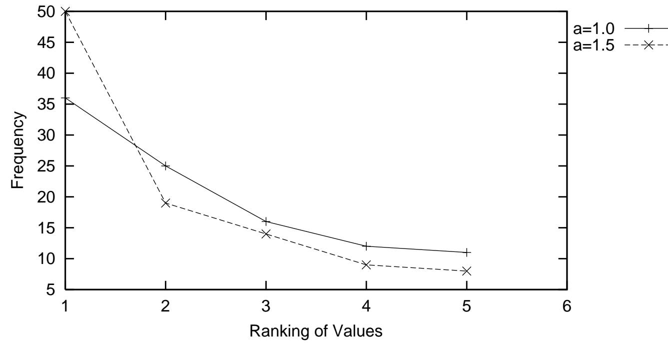

Figure 6.5: Zipf Data Distribution.

$a = 1.0$.

6.2.1 Evaluating the Size of the Context Tree

With the first experiment, we show that the mapping of the context parameters to levels is more efficient, as concerns the size of the context tree, when the context parameters with higher cardinalities in their domains are placed on the higher levels of the tree, i.e., lower at the tree. In this experiment, we count the total number of cells in the tree. Note that we count the number of cells, instead of counting the number of nodes in the context tree, because the size of each node may be different depending on the number of different values of context parameters that appear in the submitted queries.

For a context tree with three parameters, we call *ordering 1* the ordering of the context parameters when the domain of the parameter of the first level has 5 values, the next domain has 5 values too, and the last one has 50 values. *Ordering 2* is the ordering when the domains have 5, 50, 5 values respectively, and for the *ordering 3* the domains have 50, 5, 5 values. 10% of the query values are selected to be the *any* value. The rest 90% of the values are selected from the corresponding domain, either using a *uniform* data distribution, or using a *zipf* data distribution. We use the *zipf* distribution with $a = 1.0$ and $a = 1.5$. We performed this experiment 50 times for each combination of ordering and data distribution for 50 to 200 queries with a step of 50 queries. Thus, in Fig. 6.6 we can see that using a uniform distribution, the total storage space is minimized when the parameter with 50 values in its domain is assigned to the last level of the tree. In this case, the total number of cells is smaller. The results are similar when we use the zipf distribution. In particular, in Fig. 6.7, we see that the number of cells is smaller, comparing this number with the previous one,

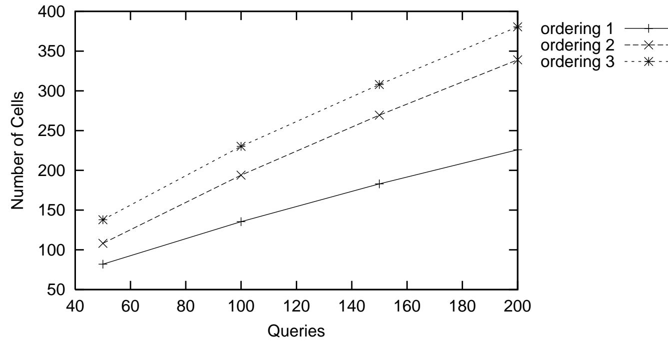

Figure 6.6: Uniform Data Distribution

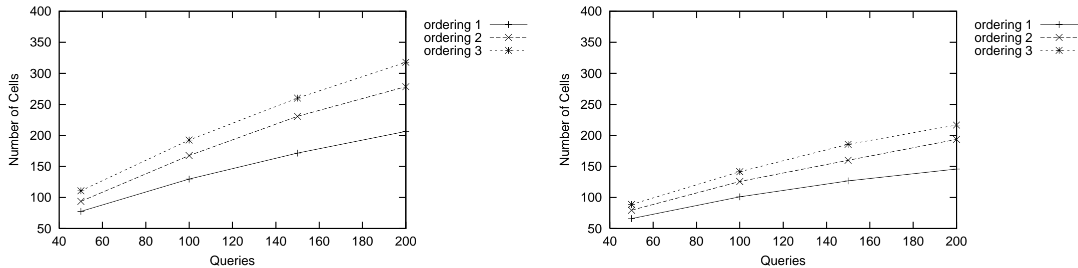

Figure 6.7: Zipf Data Distribution with $a = 1.0$ (left) and $a = 1.5$ (right)

because using the zipf distribution the "hot" values are more frequent in queries, i.e., more values are the same. In Fig. 6.7 (left) the parameter a has value 1.0, and in Fig. 6.7 (right) a has value 1.5. This large value for a, eliminates further the number of cells, because too many query values are the same.

6.2.2 Evaluating the Accuracy of Approximate Results

With the second experiment, we show how accurate are the *top-k* results of a preference query, using previous results of a relevant query. In particular, there are many cases that there is no need to compute the answer from scratch. So, we can avoid computing the answer for a query similar with a previous one, paying the cost of loosing some accuracy in the results.

We performed this experiment using the results of the previous one as concerns the mapping of the context parameters to the levels of the tree. Thus, the first and the second levels take values from the *small* domain and the third from the *large* domain. Furthermore, we use initially the *neighborhood approximation threshold* and then, the *approximation coverage threshold*. We run our

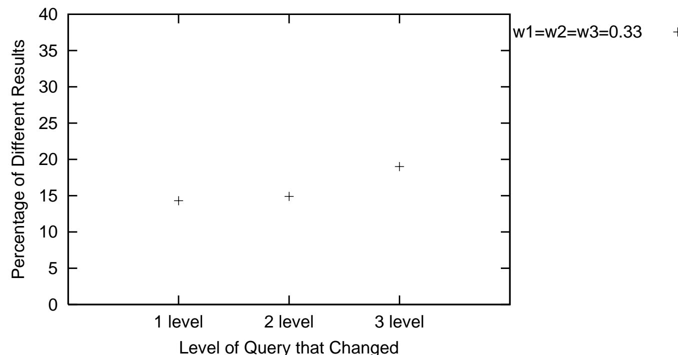

Figure 6.8: Different Results between two similar Queries.

experiments 100 times for each case, using integer discrete values. Also, the number of tuples is 1000.

Using the Neighborhood Approximation Threshold

In this set of experiments, we examine how accurate are the *top-k* results, when we do not compute them but we use previous results to produce them. That happens for a query similar with a previous one, expressing the similarity by closely related values. In order to express when two values are closely related, we use the *neighborhood approximation threshold*.

While a query is mapped to a context state, we consider at first that a query is similar with another one, when they have the same values for all the context parameters except one. This difference between the values of the same context parameter for two queries is expressed by nearby values. More specifically, we suppose that a value v_i that belongs to a small domain has as closely related the value $v_i + 1$, if v_i is odd. If v_i is even, has as closely related the value $v_i - 1$. For instance, the above means that 4 has as closely related the value 3 and respectively, 1 has as closely related the value 2. The closely related values for a value v'_i that belongs to the large domain are included in the range $[\lfloor v'_i/5 \rfloor * 5+1, \lceil v'_i/5 \rceil * 5]$, if $v'_i \ mod \ 5 \ != 0$. Otherwise, the range is $[(v'_i/5-1)*5+1, v'_i]$. That means, that the values that included for example in the range $[6, 10]$ or $[21, 25]$ are closely related. The above similarity between two queries can be generalized for queries whose corresponding context states have more than one different and nearby values.

In order to examine the accuracy of approximate results, we use the properties proved in Section 5.3. These properties express the fact that "small" changes in context values lead to "small" changes in the rating of the queries'

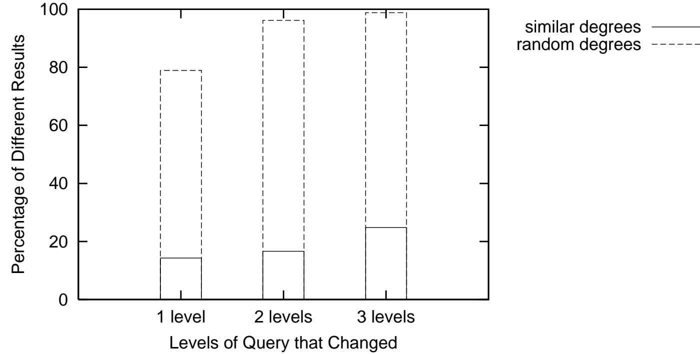

Figure 6.9: Different Results between two similar Queries with or without similar Degrees.

results. More specific, let t_1, t_2 be two tuples that have aggregate scores d_1, d_2 in a context state s and d'_1, d'_2 in a context state s' respectively. When s, s' are similar, i.e., $|d_{it} - d'_{it}| \leq \varepsilon'$, and only one of the degrees of the context parameters is changed, if $|d_1 - d_2| \leq \varepsilon$ then, $|d'_1 - d'_2| \leq \varepsilon + 2 * w_1 * \varepsilon'$.

In Fig. 6.8, queries are used whose context states have different values in one context parameter. With this experiment, we show that the percentage of different results in the *top-k* list between two similar queries is independent of the context parameter whose value is changed. Here, we used the weights $w_1 = 0.33$, $w_2 = 0.33$, and $w_3 = 0.33$. Also, the degrees of interest between two relative values differ at most $\varepsilon' = 0.08$.

In Fig. 6.9, we show that the percentage of different results in the *top-k* list between similar queries is not random, but depends on the parameter ε' (this can be understood from the above property). So, in this figure we compare the following two cases. In the first case, the degrees of interest between two similar values differs at most $\varepsilon' = 0.08$. In the second one, the degrees of closely related values are independent (selected randomly). In both cases, the weights are $w_1 = 0.33$, $w_2 = 0.33$, and $w_3 = 0.33$.

Next, we use three values for the parameter ε': 0.04, 0.08, and 0.12. Also, the weights have the values 0.5, 0.3, and 0.2. We compare first, the number of different results between two similar queries that differ at the value of one context parameter. This context parameter has a weight w_1 that is 0.5, 0.3, and 0.2, respectively. We run this experiment for each value of the parameter ε', and the results are depicted in Fig. 6.10. Then, we examine the case in which the values of two context parameters are different between the similar queries (Fig. 6.11). In this case, using the property that generalize the above one, note that the results depend on both the weights that correspond to the

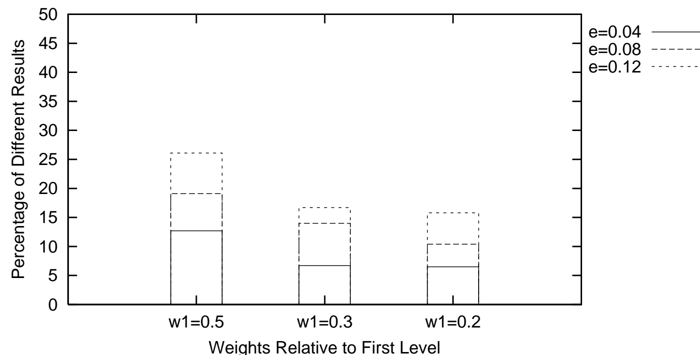

Figure 6.10: Different Results between two similar Queries when $\varepsilon' = 0.04$, $\varepsilon' = 0.08$, and $\varepsilon' = 0.12$.

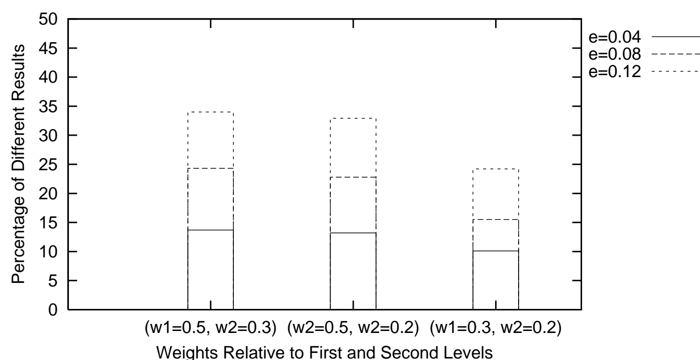

Figure 6.11: Different Results between two similar Queries when $\varepsilon' = 0.04$, $\varepsilon' = 0.08$, and $\varepsilon' = 0.12$.

context parameters that their values are changed. Finally, we study the results, when all the values are different and closely related (Fig. 6.12).

With this experiment, we show that the smaller the value of the parameter ε', the smaller the difference between the results in the *top-k* list of two similar queries. Note further, that the value of the weight that is relative to the value of the context parameter that is changed affects the previous number of different results. Thus, in a similar way, the smaller the value a weight of a context parameter takes, the smaller the number of different results.

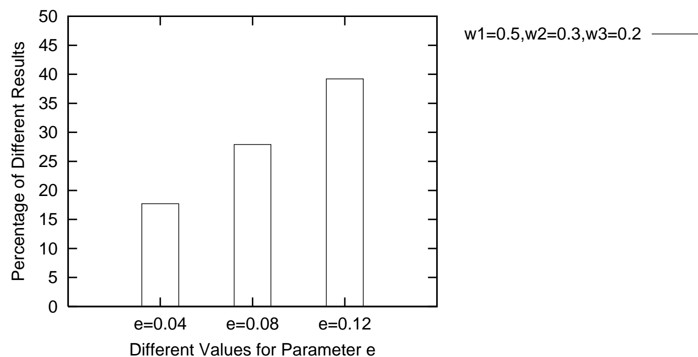

Figure 6.12: Different Results between two similar Queries when $\varepsilon' = 0.04$, $\varepsilon' = 0.08$, and $\varepsilon' = 0.12$.

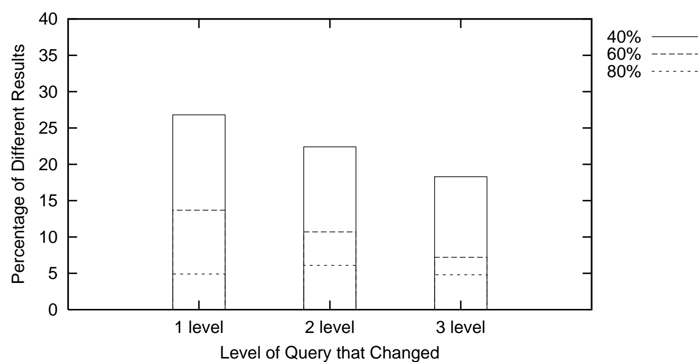

Figure 6.13: Different Results when the *Approximation Coverage Threshold* has the values 40%, 60%, 80%.

Using the Approximation Coverage Threshold

Another case where the *top-k* results are derived from a combination of other relative results is when a query contains the '*' operator, i.e., the *any* value. In this set of experiments, we use the *approximation coverage threshold* that refers to the percentage of the values of a context parameter that is expressed with the *any* value at the query, that must exist in the context tree.

We run our experiments giving to the parameter ε' the value 0.08, while the weights take the values 0.5, 0.3, and 0.2. Additionally, we use three values for the *approximation coverage threshold* namely, 40%, 60%, and 80%. Note further, that an approximation coverage threshold of k% means that at least k% of the required values are available, i.e., there are already computed and

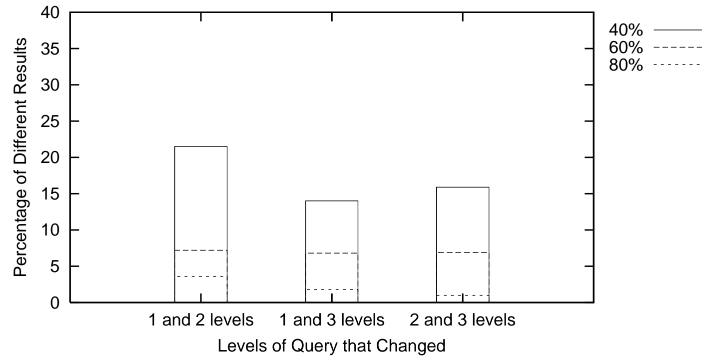

Figure 6.14: Different Results when the *Approximation Coverage Threshold* has values 40%, 60%, 80%.

stored in the context tree.

In Fig. 6.13, we present the percentage of different results in the *top-k* list, when the '*' operator is placed at the first, second and third level, for each threshold value. In Fig. 6.14, there are two '*' values, at the first and second levels, at the first and third levels, and at the second and third levels, respectively.

In both cases, the approximation is better when the '*' value is placed at the third level, which is the level that takes values from the *large* domain. That happens because in this case, more paths of the context tree are included to the relative paths, and so, more *top-k* lists of results are merged to produce the new *top-k* list.

Chapter 7

Related Work

7.1 Context-Awareness

7.2 Infrastructures for Context

7.3 Context-aware DBMS

7.4 Context-Management

7.5 Top-K Querying

In this chapter, related work is presented. First, we present infrastructures for context, which are middleware systems that address issues common to all applications that want to take advantage of context. Then, we discuss how context can be integrated into a Database Management System. Since there are many types of context information, a unifying model for modeling and a general approach for storing context parameters are challenging issues. In that way, we provide a survey of various approaches to both problems. Furthermore, in a different research topic, we present related work for determining the *top-k* results for a query.

7.1 Context-Awareness

A system is *context-aware* if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task. Each context-aware application may support one the following features [1]:

- presentation of information and services to a user;
- automatic execution of a service for a user; and
- tagging of context to information to support later retrieval.

There are various types of context-aware applications depending on the way context is used. A nice classification is provided in [27] which identifies the following categories of context-aware applications:

- *Proximate selection* is a user-interface technique where the objects located nearby are emphasized or otherwise made easier to choose.
- *Automatic contextual reconfiguration* is the process of adding new components, removing existing components, or altering the connections between components due to context changes. Typical components and connections are servers and their communication channels to clients. However reconfigurable components may also include loadable device drivers, program modules and hardware elements.
- *Contextual information and commands* can produce different results according to the context in which they are issued.
- *Context-triggered actions* are simple IF-THEN rules used to specify how context-aware systems should adapt. There are similar to contextual information and commands, except that context-triggered action commands are invoked automatically

7.2 Infrastructures for Context

Infrastructures for context are middleware systems that address issues common to all applications that want to take advantage of context. Such issues include capturing, accessing and storing context. Efficient distribution and support for independent execution from applications are desirable features for such architectures. Some popular context infrastructures include the following.

Context Toolkit. The Context Toolkit [28] is a distributed architecture that supports It is based on three abstractions: context widgets, context interpreters and context aggregators. A context widget acquires a certain type of context information and makes this information available to applications. A context interpreter accepts one or more types of context and produces a single piece of context. A context aggregator aggregates or collects context. It is responsible for all the context for a single entity. Applications can subscribe to pre-defined aggregators and supply appropriate filters.

Solar. The Solar system [29, 30] advocates a graph-based abstraction for context aggregation and dissemination. Context information is modeled through events which are produced by sources. Events flow through a directed acyclic graph of events-processing operators and are delivered to subscribing applications. Applications subscribe by describing their desired event stream as a tree of operators that aggregate low-level context information published by existing sources into the high-level context information needed by the application.

Cooltown. The Cooltown project [31] proposed a web-based model of context in which each entity (person, place or thing) has a corresponding description that can be retrieved via a URL. Using URLs for addressing, physical URL beaconing and sensing of URLs for discovery, and localized web servers for directories, they create a location-aware but ubiquitous system to support nomadic users. On top of this infrastructure, the Internet connectivity is used to support communications services.

CoolAgent. CoolAgent [32] is a context-aware multi-agent system. Ontology sharing, sensing and reasoning is supported through the use of the Resource Description Framework (RDF) and a Prolog-based system.

7.3 Context-Aware DBMS

In this section, we consider how context can be integrated into a Database Management System. Context information related to a DBMS includes user-related information (such as information provided through a user profile), computational-related (such supporting small device, limited energy, quality of network connection (e.g., in terms of reliability, frequent disconnections, intermittent connectivity an low bandwidth) and environmental conditions (weather, location, time of the day).

7.3.1 Context-Aware Query Processing

Context-aware query processing has many aspects. We consider how context affects (i) the results returned by a query, (ii) query optimization and (iii) the way the results are presented to the users.

Although, there is some research on location-aware query processing, integrating other forms of context in query processing is a new issue. The only related work that we are aware of is the context-aware query processing framework of [33]. In this framework, context-aware query processing is divided into three-phases: query pre-processing, query execution and query post-processing.

Query pre-processing is performed in two steps: a query refinement and a context binding step. The goal of the query refinement step is to further constraint the query condition by means of different contextual information. Context binding instantiates with exact values the contextual attributes involved in the refined query. After query execution, at the query post-processing phase, the results are sorted. External services may then be invoked for the delivering of results to the users. Five context-aware strategies are defined. Strategy 1 refers to queries that consider the current value of context as their reference point, for example such queries include looking for the closest restaurant, the next flight, the shortest route. To implement them, the contextual attributes are bound to their current values. Strategy 2 includes queries that access facts about the past (i.e., history data) which are recalled based on the relevant context. In this case, archived data are linked based on their common contextual attributes. Strategy 3 considers context as an additional constraint to the query. A given query is refined to include relevant constraint rules. Strategy 4 reduces the result set by ordering the produced results based on the user profile. This is achieved by using an associated sorting rule. Strategy 5 considers the delivery and presentation of results to the user by observing related delivery rules. This framework is orthogonal to our approach and a potential extension of our work includes enriching our model with constrains involving context attributes.

In the following, we shall use as a simple running example a database of information about restaurants. The type of the record entries for restaurants are tuples of a relation schema *Restaurant(id, type-of-food, address, outdoors, opening-hours, price)*. The context parameters are weather, location, time and the user profile.

Context-Aware Results

Context may affect the results produced by a query. In this case, the same query may produce different results depending on the context in which it is executed. Context-aware query processing may be seen as a two-step process. In the first phase, the context relevant to a specific query is initially identified and then acquired. During the second phase, the relevant context is integrated within the query.

Querying Context Parameters. One way to involve context within queries is by allowing explicit access to context parameters within a query. Context parameters are treated as attributes of a virtual relation; let us call this relation Context. The attributes of Context are bounded to the current value of context when the query is executed. In the following example, we assume that

the attribute time of Context is bound to the current time when the query is executed. The query returns all restaurants that are currently open.

select Restaurant.id
from restaurants, Context
where Context.time *in* Restaurant.opening-hours

Context as a Predicate. Another way to achieve context-awareness is to augment the query with appropriate predicates. In particular, a given query is transformed to a different one by adding additional constraints to it. One way this may be achieved is by adding constraints using the contextual attributes. Another way is by associating rules with specific attributes or relations and adding these rules to the query. For example assume that a user specifies that when the weather is good, the user likes to eat outdoors. The following example returns all restaurant with an outdoor facility when the weather is good and any restaurant otherwise.

An initial query submitted by the user:

select Restaurant.id
from Restaurants

is transformed to

select Restaurant.id
from Restaurants, Context
where (Context.weather = "sunny" and Restaurant.outdoors = "available") or Context.weather = "rainy"

Context as Preference. Context can be used to confine database querying by selecting as results the best matching tuples. This can be achieved by defining preferences based on context, so that under a specific context a tuple is preferred over another. In our approach, each user expresses his preference by providing a numeric score between 0 and 1. This score expresses a degree of interest. Furthermore, we divide preferences into basic (concerning a single context parameter) and aggregate ones (concerning a combination of context parameters).

Context for Associative Recall. Finally, context can be used for associative recall of past events. For example, in a "memory" database, context (such as time or location) can be used to retrieve the associated facts. For instance, it may be easier to retrieve facts (such as who was the prime minister of Greece) or objects (such as for example photographs or favorite music albums) by referring

to the particular time period of one's life (e.g., when user "John" was dating "Mary") or a geographic location (e.g., during one's vacation in Hawaii) closely associated with the facts or objects. To achieve such retrieval, storage of facts or objects in a database must include information about the context parameters when they occurred.

Context-Aware Query Optimization

Besides affecting the results of a query, context information may be used in query optimization to achieve more cost-effective plans. Computing context is very relevant in this case. Instead of optimizing disk access, query plans may be derived to optimize other performance metrics such as energy (when energy power is an issue). Furthermore, the user context can also be exploited. For instance, query processing may be such that the most relevant results (based on the user's profile) are returned first.

Context-Aware Query Presentation

The way the results are presented to a user directly depends on the device currently used by the user. For instance, when a user is interested in receiving pictures, if he uses a PDA receives pictures of lower resolution than when using a PC. In addition, energy and networking considerations may affect the way query results are delivered to the user.

7.3.2 Architecture of Context-Aware DBMS

A overview of how context can be integrated within a Database Management System (DBMS) is depicted in Figure 7.1. The *Awareness Modules* communicate with the sources that produce data (for instance, temperature sensors) and propagate any updates to the Context Manager. The *Context Manager* is responsible for managing (modeling, storing, updating) any context related information. The Context Repository is the module where context is stored. There are two ways of integrating context in a DBMS: (a) the context manager may be part of the DBMS or (b) the context manager may be seen as an intermediate middleware layer.

7.4 Context Management

There are many types of context information thus providing a unifying model for modeling and a general approach for storing context parameters are challenging issues. We provide next a survey of various approaches to both problems.

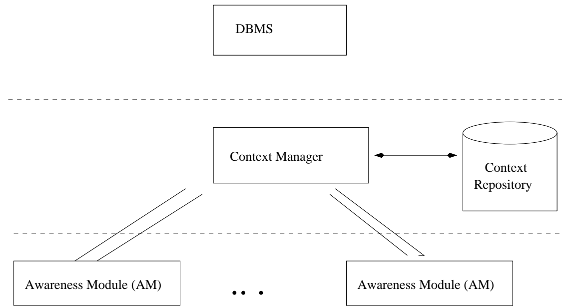

Figure 7.1: Connecting Context and Databases

7.4.1 Model of Context

A variety of models have been introduced for context. Discussions of the different models can be found in [2, 34]. Such models fall in one of the categories described next.

Models for Location. Location is a context parameter that has attracted a lot of attention. Models for location are different than other values of context mainly because the location of moving objects is a parameter whose value changes continuously with time. There are basically two different way to represent location: a symbolic and a geometric model. With the symbolic model, location is represented using abstract symbols, while with the geometric model, location is represented using coordinates. A nice overview of current research on the topic can be found in [35].

Key-Value Models. The simplest model is to represent contextual information in the form of (context-variable, value) pairs. Key-value pairs can be used for efficient exact match queries for example for automatic contextual reconfiguration. Such models are general and easy to manage but lack in expressibility of semantic information.

Markup Schemes. Context is modeled using "contextual" tags. Common to such schemes is a hierarchical data structure that is express through the nesting of tags.

An example of such representation is a CC/PP profile [36]. A CC/PP profile is a description of device capabilities and user preferences that can be used to guide the adaptation of content presented to that device. CC/PP is based on RDF, the Resource Description Framework, which was designed by the W3C as a general purpose metadata description language. The Resource Description

Framework (RDF) is used to create profiles that describe user agent capabilities and preferences. A CC/PP profile contains a number of CC/PP attribute names and associated values that are used by a server to determine the most appropriate form of a resource to deliver to a client. It is structured to allow a client to describe its capabilities by reference to a standard profile, accessible to an origin server or other sender of resource data, and a smaller set of features that are in addition to or different than the standard profile. A set of CC/PP attribute names, permissible values and associated meanings constitute a CC/PP vocabulary.

An example of using a markup scheme is "stick-e" notes [37] which are the electronic equivalents of post-it notes. Context information is modeled as tags and corresponding fields. The stick-e fields can recursively contain other tags and corresponding fields. The note of the <body> tag is automatically triggered when the contextual constraints in the <require> tag are met. This model has evolved into the ConteXtML model which is an XML-based protocol for exchanging contextual information.

Graphical Models. A variety of general models (such as the E/R model and UML) are graphical. Such models are very expressive and are mainly used as conceptual models. However, they convey little information at the instance level or on implementation issues.

Object-Oriented Models. Important features of object-oriented models are encapsulation and re-usability.

Logic-Based Models. A large number of proposals to represent context are based on logic. Important in this respect is the formalization proposed in [38]. Contexts are considered as first class objects. The basic relation is ist(c,p). It asserts that the proposition p is true in the context c. The most important formulas relate the propositions true in different contexts. Introducing contexts as formal objects permits axiomatizations in limited contexts to be expanded to transcend the original limitations. This seems necessary to provide AI programs using logic with certain capabilities that human fact representation and human reasoning possess. Fully implementing transcendence seems to require further extensions to mathematical logic, i.e., beyond the non-monotonic inference methods.

Ontology-Based Models. Ontologies have currently attracted much attention for specifying concepts and interrelations.

7.4.2 Storing Context

An important issue is what is an appropriate model for storing context. Besides storing the current context for building context-aware systems and applications, there is growing effort to extract interesting knowledge (such rules, regularities, constraints, patterns) from large collections of context data.

Storing context data using data cubes, called context cubes, is proposed in [17] for developing context-aware applications that use archive sensor data. The context cube provides a multidimensional model of context data where each dimension presents a context dimension of interest. The context cube also provides a number of tools for accessing, interpreting and aggregating context data by using concept relationships defined within the real context of the application. The basic cube operations are slice, dice, roll-up and drill-down. A slice is a selection of one dimension of an n-dimensional cube. The dice is a selection applied to all dimensions of the cube. Roll-up generates a new cube by applying an aggregate function on one dimension. Drill-down is the inverse of roll-up; it generates a context cube with finer granularity on one of the n dimensions. So in this work, data cubes are used to store historical context data and to extract interesting knowledge from collections of context data. Furthermore, a cube can be used to create new context from analysis of the existing data. In our work, we use data cubes for storing context-dependent preferences and answering related queries.

7.4.3 Updating Context

Since context parameters change with time, deriving a model of how they change is important. This enables the prediction of future values of context. Furthermore, such information can be used to fine-tune various system-related parameters as well as a variety of protocols. Finally, having a model for context updates allow building systems that are more cost-effective. There are in general two ways for communicating updates: a push and a pull model. In the *push model*, the source of the context update push the new value of the associated context parameter to the context-aware system. In the *pull model*, the context-aware system polls the source to learn about any updates. In both models, there is cost associated with update propagation. A model of context would reduce such cost, since it will eliminate the cost of communication between the source and the context-aware system. It will also reduce the computation cost at both ends.

Deriving general models for context updates is a formidable task, because of the great variety of context information. Models have been advanced for location updates, since it is possible to predict future locations when the moving objects

follow some pattern of movement or are moving in trajectories (for example, cars in highways).

Another important issue relevant to data engineering is how to communicate the change of context to the result of querying processing. We distinguish the following three different approaches regarding how to update the query so that it takes into account context:

- each query answer includes the valid time of the answer
- the previous results is cached and used them to prune the search for the new results
- the result is precomputed by using some model to predict the values of the context parameters.

7.5 Top-K Querying

In our work at the leaf nodes of the context tree, we store a list of ids, e.g. restaurant ids, along with their aggregate scores for the associated context state that is, for the path from the root leading to them. Instead of storing aggregate values for all non-context parameters, to be storage-efficient, we just store the *top-k* ids (keys) that are the ids of the items having the *k-highest* aggregate scores for the path leading to them. The motivation is that this allows us to provide users with a fast answer with the data items that best match their query. Only if more than *k-results* are needed, additional computation will be initiated. The list of ids is sorted in decreasing order according to their scores.

In [21], algorithms are given for determining the *top-k* objects for a query's result. The authors assume that each database consists of a finite set of objects. Each object has m fields $x_1, x_2, \ldots, x_m$, where $x_i \in [0, 1]$ for each i. Each x_i is a grade of an object R under one of the m attributes, and $t(x_1, x_2, \ldots, x_m)$ is the overall grade of object R for an aggregation function t. The database is thought of as consisting of m sorted lists $L_1, L_2, \ldots, L_m$. Each entry of L_i is of the form (R, x_i), where x_i is the ith field of R. Each list L_i is sorted in decreasing order by the x_i value. Also, they consider two modes of access to data: the sorted access and the random one. A *sorted access* is a sequential access from the top. Here, the system obtains the grade of an object in one of the sorted lists by proceeding through the list sequentially from the top. Thus, if object R has the lth highest grade in the ith list, then l sorted access to the ith list are required to see this grade under sorted access. In *random access*, the system requests the grade of object R in the ith list, and obtains it in one random access.

Instead of executing the naive algorithm to obtain the *top-k* answers (look at every entry in each of the m sorted lists, computing using t the overall grade of every object and return the *top-k* answers), several algorithms have been proposed. At first, Fagin [20] introduced an algorithm, named FA (Fagin's Algorithm). Initially, the FA executes sorted access to each of the m sorted lists L_i in parallel, i.e., access the top member of each of the lists, then the second member and so on. FA waits until there is a set of at least k objects, such that each of these objects has been seen in each of the m lists. FA finds the ith field of x_i, for each object with a random access to each list L_i. Finally, computes the overall grades according to the aggregation function t for all objects that have been seen and returns the objects with the k highest grades.

The Threshold Algorithm (TA) executes sorted access in parallel to each of the m sorted lists L_i. For each object R is seen executes random access to the other lists to find the grade x_i of R in every list L_i, and then computes the overall grade of R. For each list L_i, let x_i be the grade of the last object seen under sorted access. TA computes a threshold value r to be $t(x_1, \ldots, x_m)$. The algorithm stops when at least k objects have been seen whose grade is at least equal to r and returns the k objects with the highest grades. An interesting variation of TA algorithm is the approximation algorithm TA$_\theta$. This algorithm stops when at least k objects have been seen whose grade, when multiplied by θ ($\theta > 1$), is at least equal to r.

Furthermore, a No Random Access Algorithm (NRA) is proposed for systems where random accesses are forbidden. NRA executes sorted access in parallel to each of the m sorted lists L_i. At each depth d (when d objects have been accessed under sorted access in each list) the bottom values $x_1^{(d)}, x_2^{(d)}, \ldots, x_m^{(d)}$ encountered in the lists. For every object R NRA computes the lower bound $W^{(d)}(R)$ and the upper bound $B^{(d)}(R)$. The lower bound for an object R at depth d is the grade of the aggregate function t where for each unknown grade x_i we put 0. In the computation of the upper bound for each unknown grade x_i we put the value x_i, where x_i is the smallest value obtained via sorted access in list L_i. The algorithm maintains the k objects with the largest $W^{(d)}$. If two objects have the same $W^{(d)}$ values, the object with the highest $B^{(d)}$ value wins. NRA stops when k distinct objects have been seen and all the other objects have an upper bound value less or equal with the lower bound value of the kth object. A modification of the NRA algorithm is the combined algorithm CA that uses random accesses and takes their costs (relative to sorted access) into account. The main idea of CA is to run NRA, but every $h = c_R/c_S$ steps, where c_R is the cost of a sorted access and c_S the cost of a random access, to run a random access phase and update the upper and lower bounds.

Additionally, there is some similarity with the work done in the context of the

PREFER system [22] for processing *ranked queries* that are, queries that return the top objects of a database according to a preference function. The focus of this work is on a different topic. In particular, this approach precomputes materialized ranked views, in order to guarantee better query performance, and then answers a ranked query q with a preference function f', from a materialized ranked view v that is based on another preference function f, when for example the two functions f and f' have different weight values. In that way, at first the appropriate view is selected, finding which view has the maximum coverage area with the query. In following, the algorithm takes the n first objects of the view or more specific the n first objects that have values less than a threshold value. Then, reorders these objects according to the preference function of the query and returns all of them that are before the first object of the previous ordering, i.e., the ordering that based on the view's preference function. This is the output of the ranked query.

Relevant in this respect is the research in [39, 40]. In *top-k* queries [39], users specify target values for certain attributes, without requiring exact matches to these values in return. Instead, the result to such queries is typically a rank of the "top-k" tuples that best match the given attribute values. The *skyline* [40] is defined as those tuples of a relation that are not dominated by any other tuple. A tuple dominates another tuple if it is as good or better in all dimensions and better in at least one dimension.

Chapter 8

Conclusions and Future Work

The use of context is important in many applications such as in pervasive computing where it is important that users receive only relevant information. In this thesis, we consider integrating context with query processing, so that when a user poses a query in a database, the result depends on context. In particular, each user indicates preferences on specific attribute values of a relation. Such preferences depend on context. Users express their preferences on specific database instances based on a single context parameter. Such *basic preferences*, i.e., preferences associating database relations with a single context attribute, are combined to compute *aggregate preferences* that include more than one context parameter. We store basic preferences in data cubes and show how OLAP techniques can be used to compute context-aware queries, that is queries whose results depend on context.

Aggregate preferences are not explicitly stored. To improve performance, we propose storing aggregate preferences computed as results of previous queries using an auxiliary data structure called *context tree*. A path in the context tree corresponds to an assignment of values to context parameters, that is, to a context state, for which the aggregate score has been previously computed.

The context tree provides an efficient way to retrieve the *top-k* results that are relevant to a preference query. When a query is posed to the system, we first check if there exists a context state that matches it in the context tree. If so, we retrieve the *top-k* results from the associated leaf node. Otherwise, we compute the answer and insert the new context state, i.e., the new path and the associated *top-k* results, in the tree. Furthermore, the results stored in the

context tree can be re-used to speed-up query processing. This happens for a similar query, i.e., for a query that has similar values to its context state with a previous one. In this case, the results of the new query are approximate. We also show how search in the context tree can be improved using a variation of a Bloom-based filter for testing membership in the tree.

Finally, we demonstrate the feasibility of our approach through a prototype application regarding a context-aware restaurant guide, and we evaluate the performance of the context tree on answering queries.

This work is a first step towards using the context tree. There are many issues that need further investigation. So far, we used point queries. A *point query* is a simple traversal on the *context tree* structure from the root to a leaf. At level i, we search for the cell having as key the ith value of the query and descend to the next level following the appropriate pointer. If the ith value is *any*, we follow the pointer of the *any* cell.

Range queries differ from point queries because they contain at least one context parameter with a range of values. If a range is specified for the ith context parameter, i.e., for the ith level of the query, for each key satisfying the specified range we recursively descend to the corresponding subtree in a depth-first manner. In this case, the *top-k* results that are relevant to the query can be produced merging the individual results of the corresponding point queries.

We can further extend our model, as concerns the range queries, modifying the model of preferences. So far, each *basic preference* is described by a context parameter, a set of non-context parameters, and a degree of interest, while an *aggregate preference* is expressed by a set of context parameters, a set of non-context parameters, and similarly has a degree of interest. In an extended approach, users can express their basic preferences giving scores that refer to a range of values of the corresponding context parameter. Respectively, a context parameter of an aggregate preference can refer to a range of values, because each aggregate preference is derived from a combination of basic ones.

The context tree is used to cache context states that are related with previous submitted queries. We store to each context state the corresponding to the query *top-k* results, so that these results can be re-used by subsequent queries. While the context tree takes up limited space, we consider two replacement policies for choosing which paths to delete. These policies use the *LRU* and the *LFU* algorithm, respectively. However, a context state is not selected to be deleted, some of its top-k results may be not fresh. In this way, we propose to delete the old paths. To implement this policy, we can store at each path the time that the path is inserted in the tree and periodically to delete the paths with small such values. Alternatively, when a change in a basic score is occurred, we compute from scratch the aggregate score of the associated data object and we

examine if it needed an update to the *top-k* lists.

Another interesting issue, in a different area, is how context tree can contribute to the creation of clusters of nodes in decentralized p2p systems. In general, all the nodes of a cluster have similar interests. Usually, clusters in unstructured p2p systems are organized according to the content of data files stored at each node. Several approaches have been proposed in previous work. For instance, in the *associative overlays* ([41]), all nodes that belong to a cluster satisfy a *predicate*. This set of nodes is called a *guide rule* and all the guide rules define the network topology. Each node maintains a small list of other nodes that belong to the same guide rule. A search process in a guide rule is performed like the blind search in unstructured systems.

In a similar way, *Semantic Overlay Networks* (SONs - [42]) consist of clusters of nodes. Each cluster includes nodes that are semantically related. Two nodes are semantically related when the content of their data files are similar. All connections are between nodes that belong to the same SON, without the need that in a SON all nodes are connected to each other. Furthermore, a node might belong to more than one SON. Queries are processed first by finding the appropriate SON to answer it. Then, the query is propagated to this SON and finally, is performed a blind search in the specific SON. This process reduces the time to answer a query.

Another way to exploit the similarities of content of the nodes' data files is to place them (or their indices) to specific nodes. For each data file a vector is created, according to its content. This vector is used to place the data file. Also, each query has a vector. The similarity between a file's vector and a query's vector, leads the query to an appropriate node, i.e., a node that may have the result of the query. This approach is presented in [43].

In a different approach, we can use the query workload of each node and not only the content of its data files to construct clusters of nodes. In particular, the context tree can be used as an index that expresses the local query workload of a specific node. Two nodes are included in the same cluster if they have similar context trees. The similarity between two context trees refers the percentage of same paths in the trees. In this part of our future work, the multi-level Bloom filters can be used to characterize if two context trees are similar or not.

Bibliography

[1] A. K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing*, 5(1), 2001.

[2] G. Chen and D. Kotz. A Survey of Context-Aware Mobile Computing Research. Dartmouth Computer Science Technical Report TR2000-381, 2000.

[3] K. Henricksen, J. Indulska, and A. Rakotonirainy. Modeling Context Information in Pervasive Computing Systems. In *Proc. of the 1st International Conference on Pervasive*, pages 167–180, 2002.

[4] E. Pitoura and G. Samaras. Locating Objects in Mobile Computing. *IEEE TKDE*, 13(4), 2001.

[5] D. Salber, A. K. Dey, and G. D. Abowd. The Context Toolkit: Aiding the Development of Context-Enabled Applications. *CHI Conference on Human Factors in Computing Systems*, 1999.

[6] G. Chen, M. Li, and D. Kotz. Design and implementation of a large-scale context fusion network. *International Conference on Mobile and Ubiquitous Systems: Networking and Services*, 2004.

[7] P. Vassiliadis and S. Skiadopoulos. Modelling and Optimisation Issues for Multidimensional Databases. In *Proc. of 12th International on Advanced Information Systems Engineering,(CAiSE 2000), Stockholm, Sweden, June 5-9, 2000*, volume 1789 of *Lecture Notes in Computer Science*, pages 482–497. Springer, 2000.

[8] P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. *International Conference on Data Engineering*, 1997.

[9] O. Wolfson, P. Sistla, S. Chamberlain, and Y. Yesha. Updating and Querying Databases that Track Mobile Units. *Distributed and Parallel Databases Journal*, 7(3), 1999.

[10] R. Agrawal and E. L. Wimmers. A Framework for Expressing and Combining Preferences. In *Proc. of SIGMOD*, 2000.

[11] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic Ranking of Database Query Results. *International Conference on Very Large Data Bases*, 2004.

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

[13] J. Chomicki. Preference Formulas in Relational Queries. *TODS*, 28(4), Dec 2003.

[14] W. Kiessling. Foundations of Preferences in Database Systems. *International Conference on Very Large Data Bases*, 2002.

[15] W. Kiessling and G. Koestler. Preference SQL - Design, Implementation, Experiences. *International Conference on Very Large Data Bases*, 2002.

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

[17] L. Harvel, L. Liu, G. D. Abowd, Y-X. Lim, C. Scheibe, and C. Chathamr. Flexible and Effective Manipulation of Sensed Contex. In *Proc. of the 2nd Intl. Conf. on Pervasive Computing*, 2004.

[18] Y. Roussos, Y. Stavrakas, and V. Pavlaki. Towards a Context-Aware Relational Model. *In the proceedings of the International Workshop on Context Representation and Reasoning (CRR'05)*, 2005.

[19] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1), 1997.

[20] R. Fagin. Combining Fuzzy Information from Multiple Systems. *Journal of Computer and System Sciences*, 1999.

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

[22] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *The VLDB Journal*, 13(1):49–70, 2004.

[23] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the PetaCube. In *Proc. ACM SIGMOD*, 2002.

[24] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[25] G. Koloniari and E. Pitoura. Bloom-Based Filters for Hierarchical Data. *5th Workshop on Distributed Data Structures and Algorithms*, 2003.

[26] G. Koloniari and E. Pitoura. Filters for XML-based Service. *Discovery in Pervasive Computing*, 47(4):461–474, 2003.

[27] B. N. Schilit, N. I. Adams, and R. Want. Context-Aware Computing Applications. In *Proc. of the Workshop on Mobile Computing Systems and Application*, 1994.

[28] A. K. Dey. Providing Architectural Support for Building Context-Aware Applications. PhD Thesis, College of Computing, Georgia Institute of Technology, December 2000, 2000.

[29] G. Chen and D. Kotz. Solar: An Open Platform for Context-Aware Mobile Applications. In *Proc. of the 1st International Conference on Pervasive Computing*, 2002.

[30] G. Chen and D. Kotz. Context Aggregation and Dissemination in Ubiquitous Computing Systems. In *Proc. of the 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA02)*, 2002.

[31] T. Kindberg, J. Barton, and J. Morgan et al. People, Places, Things: Web Presence for the Real World. *MONET*, 7(5), 2002.

[32] H. Chen, S. Tolia, C. Sayers, T. Finin, and A. Joshi. Creating Context-Aware Software Agents. In *Proc. of the First GSFC/JPL Workshop on Radical Agent Concepts*, 2002.

[33] L. Feng, P.M.G. Apers, and W. Jonker. Towards Context-Aware Data Management for Ambient Intelligence. In *Proc. of the 15th Intl. Conf. on Database and Expert Systems Applications (DEXA)*, 2004.

[34] T. Strang and C. Linnhoff-Popien. A Context Modeling Survey. In *Proc. of the Workshop on Advanced Context Modelling, Reasoning and Management associated with the Sixth International Conference o n Ubiquitous Computing (UbiComp 2004)*, 2004.

[35] M. Koubarakis, T. K. Sellis, and A. U. Frank et. al. *Spatio-Temporal Databases: The CHOROCHRONOS Approach.* Lecture Notes in Computer Science 2520 Springer, 2003.

[36] W3C. Composite Capabilities/Preferencs Profile (CC/PP). http://www.w3.org/TR/2004/REC-CCPP-struct-vocab-20040115/, 2004.

[37] P. J. Brown, J. D. Bovey, and X. Chen. Context-aware Applications: from the Laboratory to the Marketplace. *IEEE Personal Communications*, 4(5), 1998.

[38] J. McCarthy. Notes in Formalizing Context. In *Proc. of the 13th International Joint Conference in Artificial Intelligence*, 1993.

[39] S. Chaudhuri and L. Gravano. Evaluating Top-k Selection Queries. In *Proc. of VLDB*, 1999.

[40] S. Bfrzsfnyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *Proc. of ICDE*, 2001.

[41] E. Cohen, A. Fiat, and H. Kaplan. Associative Search in Peer to Peer Networks: Harnessing Latent Semantics. *In Proceedings of the IEEE INFOCOM'03 Conference*, 2003.

[42] A. Crespo and H. Garcia-Molina. Semantic Overlay Networks for P2P Systems. Stanford University, Technical Report, http://www-db.stanford.edu/peers, 2003.

[43] C. Tang, Z. Vu, and S. Dwarkadas. Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks. *In Proceedings of the ACM SIGCOMM'03 Conference*, 2003.

Index

Approximation Coverage Threshold, 36
Associative Overlays, 70
- Guide Rule, 70

Atomic Query Element, 18
Attribute, 11
Awareness Module, 62

Bloom Filter, 42
- Breadth Bloom Filter, 43
- Depth Bloom Filter, 43
- False Positive, 42

CC/PP Profile, 63
Clusters, 70
Context, 5
- Quality of Context, 6
- Types of Context, 5
  - Computing Context, 5
  - Physical Context, 5
  - Time Context, 5
  - User Context, 5

Context Environment, 12
Context Manager, 62
Context Parameter, 12
- Dynamic Context Parameter, 13
- Static Context Parameter, 13

Context Relational Model, 25
- World, 25

Context Repository, 62
Context State, 12
Context Tree, 32
Context-Aware Application, 58
- Automatic contextual reconfiguration, 58
- Context-triggered actions, 58
- Contextual information and commands, 58
- Proximate selection, 58

Context-Aware Query, 7
Context-Aware Query Processing, 59
- Query Execution, 59
- Query Post-Processing, 59
- Query Pre-Processing, 59

Context-Aware System, 7, 57
ConteXtML, 64
Contextual Preference, 14
- Aggregate Preference, 15
- Basic Preference, 14

Cube, 21
- Cell, 21
- Dimension, 21
- Measure, 21

Domain, 11
Dwarf, 35

Hierarchies for Attributes, 12
Hypercube, 21

Infrastructures for Context, 58
- Context Toolkit, 58
- CoolAgent, 59
- Cooltown, 59
- Solar, 59

Inheriting Preference, 15

Least Frequently Used, 41
Least Recently Used, 41

Neighborhood Approximation Threshold, 37

OLAP, 21, 29
- Dice, 29
- Drill-down, 30
- Roll-up, 29
- Slice, 29

Point Query, 70
PREFER, 68
Preference Formula, 18
Preference Restaurant Guide, 45
Preference SQL, 18
- Basic Preference Type, 19
  - Around, 19
  - Highest, 19
- Complex Preference Type, 19
  - Pareto Accumulation, 19
- Hard Constraints, 18
- Soft Constraints, 19

Qualitative Preference, 18
Quality of Service, 7
- Accuracy, 7
- Conflict-free, 7
- Level of Detail, 7
- Timeliness, 7

Quantitative Preference, 17

Range Query, 70
Ranked Queries, 68
Relation, 11
Resource Description Framework, 63

Semantic Overlay Networks, 71
Situated Preference, 20
Situation, 19
- Influence, 19
  - Personal, 19
  - Surrounding, 19
- Location, 19
- Timestamb, 19

Skyline, 68
Star Schema, 22
- Dimension Table, 22
- Fact Table, 22

Top-K, 33

Uniform Data Distribution, 51
Updating Context, 65
- Pull Model, 65
- Push Model, 65

Zipf Data Distribution, 51