Adding Context to Preferences

Kostas Stefanidis, Evaggelia Pitoura and Panos Vassiliadis
Computer Science Department
University of Ioannina
GR-45110 Ioannina, Greece
{kstef, pitoura, pvassil}@cs.uoi.gr

Abstract

To handle the overwhelming amount of information currently available, personalization systems allow users to specify the information that interests them through preferences. Most often, users have different preferences depending on context. In this paper, we introduce a model for expressing such contextual preferences. Context is modeled as a set of multidimensional attributes. We formulate the context resolution problem as the problem of (a) identifying those preferences that qualify to encompass the context state of a query and (b) selecting the most appropriate among them. We also propose an algorithm for context resolution that uses a data structure, called the profile tree, that indexes preferences based on their associated context. Finally, we evaluate our approach from two perspectives: usability and performance.

1. Introduction

Today, a very large and steadily increasing amount of information is available to a wide spectrum of users, thus creating the need for personalized information processing. Instead of overwhelming the users with all available data, a personalized query returns only the information that is of interest to them [11]. In general, to achieve personalization, users express their preferences on specific pieces of data either explicitly or implicitly. The results of their queries are then ranked based on these preferences. However, most often users may have different preferences under different circumstances. For instance, a user visiting Athens may prefer to visit *Acropolis* in a nice sunny summer day and the *archaeological museum* in a cold and rainy winter afternoon. In other words, the results of a preference query may depend on context.

Context is a general term used to capture any information that can be used to characterize the situations of an entity [5]. Common types of context include the *computing context* (e.g., network connectivity, nearby resources), the *user context* (e.g., profile, location), the *physical context* (e.g., noise levels, temperature), and *time* [2]. A *context-aware* system is a system that uses context to provide relevant information and services to its users. In this paper, we consider a *context-aware* preference database system which supports preferences based on context.

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

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

We also propose an algorithm for locating those preferences that refer to the context states that are most relevant to the context states of the query. The algorithm takes advantage of a data structure, called *profile tree*, that indexes users preferences based on their associated context. Intuitively, the algorithm starts from the query context and incrementally "increases" its coverage, until a matching state is found in the profile tree. Finally, we evaluate our approach from two perspectives: usability and performance.

In summary, the main contributions of this paper are:

- We introduce a model for representing and specifying context through context descriptors that allows the specification of the context states that are relevant to preferences and queries at various levels of detail.
- We formulate the context resolution problem as the problem of identifying the context states that qualify to encompass the context state of a query and propose appropriate distance functions between context states as well as an algorithm to determine the best among them.
- We introduce the profile and the context query tree for indexing contextual preference and contextual query results respectively.

The rest of this paper is organized as follows. In Section 2, we present our reference example. In Section 3, we introduce our context and preference model and the profile tree. In Section 4, we focus on processing contextual queries, while in Section **??**, we describe the context query tree. Section 6 presents related work. Finally, Section 7 concludes the paper with a summary of our contributions.

2. Reference Example

We consider a simple database that maintains information about $points\_of\_interest$. The $points\_of\_interest$ may be for example museums, monuments, archaeological places or zoos. The database schema consists of a single database relation: *Points_of_Interest(pid, name, type, location, open-air, hours_of_operation, admission_cost)*. We consider three context parameters as relevant: $location$, $temperature$ and $accompanying\_people$. Users have preferences about $points\_of\_interest$ that have specific attribute values. Such preferences are expressed by providing a numeric score between 0 and 1 depending on the values of the context parameters.

For instance, a user may give to a value of the attribute *open-air* different scores depending on temperature, i.e., an *open-air* point_of_interest takes a lower score when the weather is $cold$ than when the weather is $warm$. Furthermore, the current user's location affects the degree of interest of a $location$ in which a $point\_of\_interest$ is placed (usually, users prefer to visit places that are nearby their current location). Similarly, the interest score of a preference that is related to the $type$ of the visiting place depends on the $accompanying\_people$ that might be $friends$, $family$, or $alone$. For example, a $museum$ may be a better place to visit than a $brewery$ in the context of $family$.

3. Context and Preference Model

Our model is based on relating context and database relations through preferences. First, we present the fundamental concepts related to context modeling, and then, we proceed in defining user preferences.

3.1. Modeling Context

Context is modeled through a finite set of special-purpose attributes, called *context parameters* (C_i). In particular, for a given application X, we define its context environment CE_X as a set of n context parameters $\{C_1, C_2, \dots, C_n\}$. For instance, the context environment of our example is {*location, temperature, accompanying_people*}. Each context parameters C_i is characterized by a *context domain*, $dom(C_i)$. As usual, a *domain* is an infinitely countable set of values.

A *context state* corresponds to an assignment of values to context parameters at some point in time. In particular, a context state w is a n–tuple of the form $(c_1, c_2, \dots, c_n)$, where $c_i \in dom(C_i)$. For instance, a context state in our example may be: $(Plaka, warm, friends)$. The set of all possible context states called *world*, W, is the Cartesian product of the domains of the context attributes: $W = dom(C_1) \times dom(C_2) \times \dots \times dom(C_n)$.

To allow more flexibility in defining preferences, we model context parameters as multidimensional attributes. In particular, we assume that each context parameter participates 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 $(L, \prec)$: $L = (L_1, \dots, L_{m-1}, ALL)$ of m *levels* and $\prec$ is a partial order among the levels of L such that $L_1 \prec L_i \prec ALL$, for every $1 < i < m$. We require that the upper bound of the lattice is always the level ALL, so that we can group all values into the single value '*all*'. The lower bound of the lattice is called the *detailed level* of the context parameter. We use the notation $dom_{L_j}(C_i)$ for the domain of level L_j of parameter C_i. For the domain of the detailed level, we shall use both $dom_{L_1}(C_i)$ and $dom(C_i)$ interchangeably.

For instance, consider the hierarchy $location$ of Fig. 1. Levels of $location$ are $Region$, $City$, $Country$, and ALL. $Region$ is the most detailed level. Level ALL is the most coarse level.

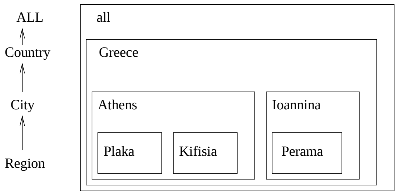

Figure 1. Hierarchies on location.

The relationship between the values of the context levels is achieved through the use of the set of $anc_{L_i}^{L_j}$, $L_i \prec L_j$, functions [19]. A function $anc_{L_i}^{L_j}$ assigns a value of the domain of L_i to a value of the domain of L_j. For instance, $anc_{Region}^{City}(Plaka) = Athens$. Formally, the set of functions $anc_{L_i}^{L_j}$ satisfies the following conditions:

1. For each pair of levels L_1 and L_2 such that $L_1 \prec L_2$, the function $anc_{L_1}^{L_2}$ maps each element of $dom_{L_1}(C_i)$ to an element of $dom_{L_2}(C_i)$.
2. Given levels L_1, L_2 and L_3 such that $L_1 \prec L_2 \prec L_3$, the function $anc_{L_1}^{L_3}$ equals to the composition $anc_{L_2}^{L_1} \circ anc_{L_3}^{L_2}$.
3. For each pair of levels L_1 and L_2 such that $L_1 \prec L_2$, the function $anc_{L_1}^{L_2}$ is monotone, i.e., $\forall$ x, y $\in dom_{L_1}(C_i)$, $L_1 \prec L_2$, $x < y \Rightarrow anc_{L_1}^{L_2}(x) \leq anc_{L_1}^{L_2}(y)$.

The function $desc_{L1}^{L2}$ is the inverse of anc_{L1}^{L2}, that is $desc_{L_1}^{L_2}(v)$= $\{x \in dom_{L_1}(C_i) : anc_{L1}^{L2}(x) = v\}$. For instance, $desc_{Region}^{City}(Athens)$ = {Plaka, Kifisia} and $desc_{City}^{Country}(Greece)$ = {Athens, Ioannina}. Finally, we use $L_1 \preceq L_2$ between two levels to mean $L_1 \prec L_2$ or $L_1 = L_2$.

Regarding our running example, levels of location are $Region$, $City$, $Country$ and ALL. For weather, there are three levels: the detailed level $Conditions$ (L_1) whose domain includes the values $freezing$, $cold$, $mild$, $warm$ and hot, the level *Weather Characterization* (L_2) which just refers to whether the weather is *good* (grouping $mild$, $warm$ and hot) or *bad* (grouping $freezing$ and $cold$) and the level ALL (L_3) so that we can group all the values into the single value 'all'. Finally, the context parameter $accompanying\_people$ has the lower level $Relationship$ (L_1) witch consists of the values $friends$, $family$, $alone$ and the level ALL (L_2). Figure 2 depicts the hierarchies on $location$, $temperature$ and $accompanying\_people$.

We define the extended domain for a parameter C_i with m levels as $edom(C_i)$ = $\cup_{j=1}^{m} dom_{L_j}(C_i)$. Then, an *extended context state* is an assignment of values to context parameters from their extended domain. In particular, an extended context state s is a n–tuple of the form

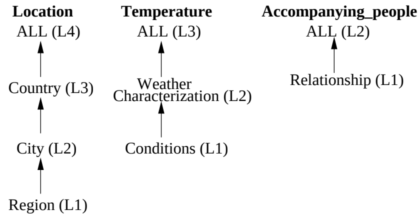

Figure 2. Hierarchies on location, temperature and accompanying_people.

$(c_1, c_2, \ldots, c_n)$, where $c_i \in edom(C_i)$. For instance, a context state in our example may be $(Greece, warm, friends)$ or $(Greece, good, all)$. The set of all possible extended context states called *extended world*, EW, is the Cartesian product of the extended domains of the context attributes: $EW = edom(C_1) \times edom(C_2) \times \ldots \times edom(C_n)$.

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

Definition 1 (Context parameter descriptor) *A context parameter descriptor $cod(C_i)$ for a parameter C_i is an expression of the form:*

1. *C_i = V, where $V \in edom(C_i)$, or*
2. *$C_i \in \{value_1, \ldots, value_m\}$, where $value_k \in edom(C_i)$, $1 \leq k \leq m$, or*
3. *$C_i \in [value_1, value_m]$, where $[value_1, value_m]$ denotes a range of values $x \in edom(C_i)$, such that $value_1 \leq x \leq value_m$.*

For example, given a context parameter $location$, a context parameter descriptor can be of the form $location = Plaka$, or $location \in \{Plaka, Acropolis\}$. Given a context parameter $temperature$, a range-based context parameter descriptor can be of the form $temperature \in [mild, hot]$, signifying thus the set of values $\{mild, warm, hot\}$.

There is a straightforward way to translate context parameter descriptors to sets of values. Practically, this involves translating range descriptors to sets of values (remember that all domains are infinitely countable, hence, they are not dense and all ranges can be translated to finite sets of values).

Definition 2 (Context of a context parameter descriptor) *Given a context parameter descriptor $c = cod(C_i)$ for a parameter C_i, its context is a finite set of values, computed as follows:*

$$Context(c) = \begin{cases} \{v\} & \textit{if c of the form } C_i = v \\ \{v_1, \ldots, v_m\} & \textit{if c of the form} \\ & C_i \in \{v_1, \ldots, v_m\} \\ \{v_1, \ldots, v_m\} & \textit{if c of the form} \\ & C_i \in [v_1, v_m] \end{cases}$$

A context descriptor is a specification that a user can make for a set of context parameters, through the combination of simple parameter descriptors.

Definition 3 (Composite context descriptor) *A (composite) context descriptor cod is a formula* $cod(C_{i_1}) \wedge cod(C_{i_2}) \wedge \ldots \wedge cod(C_{i_k})$ *where each* C_{i_j}, $1 \leq j \leq k$ *is a context parameter and there is at most one parameter descriptor per context parameter* C_{i_j}.

Given a set of context parameters $C_1, \ldots, C_n$, a composite context descriptor can describe a set of possible context states, with each state having a specific value for each parameter. Clearly, one context descriptor can produce more than one states. The production of these states can be performed by computing the Cartesian product of the context states of all the individual parameter descriptors of a context descriptor. If there is no parameter descriptor for a context parameter, then the value all is assumed. Observe, that the set of produced states is finite, due to the finite character of the context of the parameter descriptors.

Definition 4 (Context of a context descriptor) *Assume a set of context parameters* $C_1, \ldots, C_n$ *and a context descriptor cod* $= cod(C_{i_1}) \wedge cod(C_{i_2}) \wedge \ldots \wedge cod(C_{i_k})$, $0 \leq k \leq n$. *Without loss of generality, we assume that the parameters without a parameter descriptor are the last* $n-k$ *ones. The context states of a context descriptor, called* $Context(cod)$ *are defined as:*
$Context(cod(C_{i_1})) \times \ldots \times Context(cod(C_{i_k})) \times \{all\} \times \ldots \times \{all\}$

Suppose for instance, the context descriptor (location = $Plaka$ $\wedge$ temperature = $\{warm,\ hot\}$ $\wedge$ accompanying_people = $friends$). This descriptor corresponds to the following two context states: $(Plaka, warm, friends)$ and $(Plaka, hot, friends)$. In case a context descriptor does not contain all context parameters, that means that the absent context parameters have irrelevant values. This is equivalent to a condition $C_i = all$.

3.2. Contextual Preferences

In this section, we define how context affects the results of queries, so that the same query returns different results based on the context of its execution. Such context-aware personalization is achieved through the use of preferences. In particular, users express their preferences on specific database instances for a variety of context states.

In general, there are two different approaches for expressing preferences: a quantitative and a qualitative one. With the *quantitative approach*, preferences are expressed indirectly by using scoring functions that associate a numeric score with every tuple of the query answer. In the *qualitative approach* (such as the work in [4]), preferences between the tuples in the query answer are specified directly, typically using binary preference relations.

Although, our context model can be used for extending both quantitative and qualitative approaches, we use a simple quantitative preference model to demonstrate the basic issues underlying contextualization. In particular, users express their preference for specific database instances by providing a numeric score which is a real number between 0 and 1. This score expresses their degree of interest. Value 1 indicates extreme interest, while value 0 indicates no interest. Interest is expressed for specific values of non context attributes of a database relation, for instance for the various attributes (e.g., *type, location*) of our *Point_of_Interest* database relation. This is similar to the general quantitative framework of [1].

Thus, each *contextual preference* is described by (a) a context descriptor *cod*, (b) a set of values $a_1, a_2, \ldots, a_m$ of corresponding non-context parameters $A_1, A_2, \ldots, A_m$, with $a_i \in dom(A_i)$, and (c) a degree of interest, i.e., a real number between 0 and 1. The meaning is that in the set of context states specified by *cod*, all database tuples (instances) for which the attributes $A_1, A_2, \ldots, A_m$ have respectively values $a_1, a_2, \ldots, a_m$ are assigned the indicated interest score. Formally,

Definition 5 (Contextual preference) *A contextual preference is a triple of the form contextual_preference=(cod, attributes_clause, interest_score), where cod is a context descriptor, the attributes_clause* $\{A_1\theta_1 a_1, A_2\theta_2 a_2, \ldots, A_k\theta_k a_k\}$ *specifies a set of attributes* $A_1, A_2, \ldots, A_k$ *with their values* $a_1, a_2, \ldots, a_k$ *with* $a_i \in dom(A_i)$, $\theta_i \in \{=, <, >, \leq, \geq, \neq\}$ *and interest_score is a real number between 0 and 1.*

Since our focus in this paper is on context descriptors, we further simplify our model, so that in the following, we shall use $attributes\_clauses$ with a single attribute A of the form $A = a$, for $a \in dom(A)$. Further, we assume that for tuples for which more than one preference applies, appropriate combining preference functions exist [1].

In our reference example, there are three context parameters $location$, $temperature$ and $accompanying\_people$. As non-context parameters, we use the attributes of the relation $Points\_of\_Interest$. For example, consider that a user wants to express the fact that, when she is at $Plaka$ and the weather is *warm*, she likes to visit $Acropolis$. This may be expressed through the following contextual preference: $contextual\_preference_1$ = ((location = $Plaka$ $\wedge$ temperature = $warm$), (name = $Acropolis$), 0.8). Similarly, she may also express the fact that when she is with friends, she likes to visit breweries through a preference of the form: $contextual\_preference_2$ = ((accompanying_people = $friends$), (type = $brewery$), 0.9). More involved context descriptors may be used as well. As an example, consider the preference: $contextual\_preference_3$ = ((location = $Plaka$ $\wedge$ temperature $\in$ $\{warm, hot\}$), (name = $Acropolis$), 0.8), where the context descriptor is *cod* = ($location$ = $Plaka$ $\wedge$ temperature $\in$ $\{warm, hot\}$).

A contextual preference may conflict with another one. For example, assume that a user defines that she prefers to visit $Acropolis$ in a nice sunny day, giving to this preference a high score of 0.8. If, later on, she gives to the same preference the interest score 0.3, this will cause a conflict. Formally, a conflict between contextual preferences is defined as follows:

Definition 6 (Conflicting preferences) *A* $contextual\_preference_i$ *= (*cod_i*, (*A_i *=* a_i*),* $interest\_score_i$*) conflicts with a* $contextual\_preference_j$ *= (*cod_j*, (*$A_j = a_j$*),* $interest\_score_j$*) if and only if:*

1. $Context(cod_i) \cap Context(cod_j) \neq \emptyset$*, and*

2. A_i = A_j *and* a_i = a_j*, and*

3. $interest\_score_i \neq interest\_score_j$*.*

Such conflicting preferences are detected when users enter their preferences. Finally, we define $profile\ P$ as:

Definition 7 (Profile) *A* $profile$ P *is a set of non-conflicting contextual preferences.*

3.3. The Profile Tree

In this section, we present a scheme for storing contextual preferences. Contextual preferences are stored in a hierarchical data structure called *profile tree*, as shown in Fig. 3. Recall that each contextual preference is expressed using a context descriptor, *cod*, that specifies a set of context states. The basic idea is to store in the profile tree, all context states that correspond to context descriptors that appear in the profile P. Each such context state will correspond to a single root-to-leaf path in the profile tree. Then, a set of paths will constitute the context descriptor of a contextual preference. In each leaf of the tree, we store the attribute_clause and interest score applicable to the context state corresponding to the path leading to this leaf.

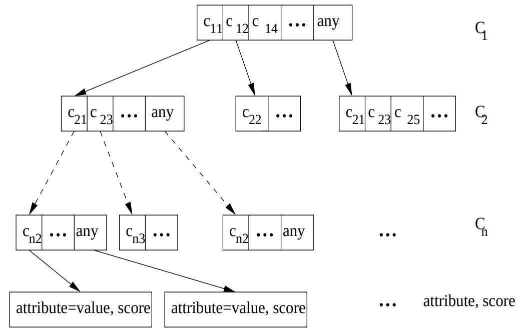

Figure 3. The profile tree.

Assume that the context environment CE_X has n context parameters $\{C_1, C_2, \ldots, C_n\}$. There is one level for each context parameter and one additional level for the leaves. Thus, the height of the tree is n + 1. Each context parameter is assigned to one level of the tree. For simplicity, assume that context parameter C_i is mapped to level i.

Let P be a profile and C_P be the set of context descriptors that appear in the contextual preferences in P. There is a path in the tree for each context state s of each *cod* in C_P. The profile tree for P is constructed as follows. Each non-leaf node at level k contains cells of the form $[key, pointer]$, where key is equal to $c_{kj} \in edom(C_k)$ for a value of the context parameter C_k that appeared in state s of a context descriptor *cod* in C_P. 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 state s of *cod*. In addition, key may take the special value all, which corresponds to the lack of the specification of the associated context parameter in *cod*. Each leaf node has the form $[attribute = value, interest\_score]$, where $[attribute = value, interest\_score]$ is the one associated with *cod*.

Any conflicting contextual preferences are detected during their insertion in the profile tree. Each contextual preference is associated with a set of paths of the profile tree, one for each of the context states of the context produced from its descriptor *cod*. When a path (i.e., state) is inserted in the tree, we check whether the same path already exists, thus leading to a conflicting preference. If this is the case, the path is not inserted and the user is notified. Note that to detect conflicts, a single traversal of a root-to-leaf path suffices.

In summary, a profile 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 levels corresponds to a context parameter and the last

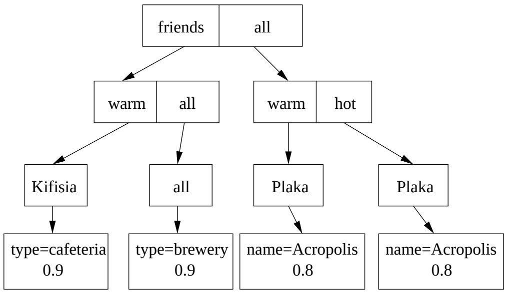

Figure 4. An instance of a profile tree.

one to the leaf nodes.

- Each non-leaf node at level k maintains cells of the form $[key, pointer]$, where $key \in edom(C_k)$ for some value of c_k that appeared in a preference or $key = all$. 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 context descriptor with the key.
- Each leaf node stores an attribute with its value and related degree of interest of the contextual preference that corresponds to the path leading to it.

For example, assume an instance of a profile P consisting of the following preferences: $contextual\_preference_1$ = ((location = $Kifisia$ $\wedge$ temperature = $warm$ $\wedge$ accompanying_people = $friends$), (type = $cafeteria$), 0.9), $contextual\_preference_2$ = ((accompanying_people = $friends$), (type = $brewery$), 0.9), and $contextual\_preference_3$ = ((location = $Plaka$ $\wedge$ temperature $\in$ $\{warm, hot\}$), (name = $Acropolis$), 0.8). Assume further that the three context parameters of our reference example are assigned to levels as follows: $accompanying\_people$ is assigned to the first level of the tree, $temperature$ to the second and $location$ to the third one. For the above contextual preferences, the profile tree of Fig. 4 is constructed.

The way that the context parameters are assigned to the levels of the context tree affects its size. Let m_i, $1 \leq i \leq n$, be the cardinality of the domain, then the maximum number of cells is $m_1 * (1 + m_2 * (1 + \ldots (1 + m_n)))$. The above number is as small as possible, when $m_0 \leq m_1 \leq \ldots \leq m_k$, thus, it is better to place context parameters with domains with higher cardinalities lower in the context tree.

4. Contextual Preference Queries

In this section, we define contextual queries. Then, we formulate the problem of identifying the preferences that are most relevant to a query and present an algorithm that locates them.

4.1. Contextual Queries

A contextual query is a query enhanced with information regarding context. Implicitly, the context associated with a contextual preference query is the current context, that is, the context surrounding the user at the time of the submission of the query. The current context should correspond to a single context state, where each of the values of the context parameters takes a specific value from its most detailed domain. However, in some cases, it may be possible to specify the current context using only rough values, for example, when the values of some context parameters are provided by sensor devices with limited accuracy. In this case, a context parameter may take a single value from a higher level of the hierarchy or even more than one value.

Besides the implicit context, we also consider queries that are explicitly augmented with an extended context descriptor. For example, a user may want to pose an exploratory query of the form: "When I travel to Athens with my family this summer (implying good weather), what places should I visit?". Formally,

Definition 8 (Extended context descriptor) *An extended context descriptor, ecod is a formula of the following form:* $(cod_{11} \wedge \ldots \wedge cod_{1j}) \vee \ldots \vee (cod_{l1} \wedge \ldots \wedge cod_{lm})$ *where* cod_{ij} *is a context descriptor.*

Definition 9 (Contextual query) *A contextual query CQ is a query Q enhanced with an extended context descriptor denoted* $ecod^Q$.

Now, the problem is: given the $ecod^Q$ of a contextual query CQ and a user profile P, identify the contextual preferences that are the most relevant to the context states specified by $ecod^Q$. Next, we first formalize the problem and then, provide a procedure for locating such preferences.

4.2. Context State of a Query

Assume a contextual query CQ enhanced with an extended context descriptor consisting of a context descriptor of the form $ecod^Q$ = $(location = Athens \wedge weather = warm)$ and a simple profile P = { (($location$ = $Greece$ $\wedge$ $weather$ = $warm$), $attributes\_clause$, $interest\_score_1$), (($location$ = $Europe$ $\wedge$ $weather$ = $warm$), $attributes\_clause$, $interest\_score_2$)}. Intuitively, we are seeking for a context descriptor in P that is more general than the query descriptor, in the sense that its context covers that of the query. Both context descriptors in P satisfy this requirement, however, the first one is more "specific" and should be the one used.

First, we formalize the notion of a set of states *covering* another one.

Definition 10 (Covering context state) *An extended context state $s^1 = (c^1_1, c^1_2, \ldots, c^1_n) \in EW$ covers an extended context state $s^2 = (c^2_1, c^2_2, \ldots, c^2_n) \in EW$, iff $\forall$ k, $1 \leq k \leq n$, $c^1_k = c^2_k$, or $c^1_k = anc^{L_j}_{L_i}(c^2_k)$ for some levels $L_i \prec L_j$.*

It can be shown that the $covers$ relationship among states is a partial order.

Theorem 1 *The $covers$ relationship among states is a partial order relationship.*

Proof: We must prove that the $covers$ relationship is (1) reflexive (i.e., s covers s), (2) antisymmetric (if s^1 covers s^2 and s^2 covers s^1, then $s^1 = s^2$) and (3) transitive (if s^1 covers s^2 and s^2 covers s^3, then s^1 covers s^3).

1. Reflexivity is straightforward.
2. Assume for the purpose of contradiction, that the antisymmetric property does not hold. In this case, there is a certain parameter k, for which, $c^1_k = anc^{L_1}_{L_2}(c^2_k)$ and $c^2_k = anc^{L_2}_{L_1}(c^1_k)$. But, this cannot happen due to the partial order of levels in a hierarchy.
3. The transitivity property is proved similarly.

Definition 11 (Covering set) *A set S_i of extended context states, $S_i \subseteq EW$ covers a set S_j of extended context states, $S_j \subseteq EW$, iff $\forall$ $s \in S_j$, $\exists$ $s' \in S_i$, such that s' covers s.*

Now, we define formally, which context descriptor matches the state of a query.

Definition 12 (Matching context) *Let P be a profile, cod a context descriptor and C_P the set of context descriptors appearing in the contextual preferences of P. We say that a context descriptor cod' $\in C_P$ is a match for cod iff*

(i) $Context(cod')$ $covers$ $Context(cod)$, and

(ii) $\neg\exists$ cod'' $\in C_P$, cod'' $\neq$ cod', such that $Context(cod')$ $covers$ $Context(cod'')$ and $Context(cod'')$ $covers$ $Context(cod)$.

There are two issues, one is whether there is at least one context preference that matches a given cod and the other one is what happens if there are more than one match. Regarding the first issue, if there is no matching context, we consider that there is no context associated with the query. In this case, the query is executed as a normal (i.e., non contextual) preference query. Note that the user can define non contextual preference queries, by using empty context descriptors which correspond to the $(all, all, \ldots, all)$ state (see Def. 4).

As an example for the case of more than one match, consider again the $ecod$ = ($location = Athens \wedge weather = warm$) and the profile P ={ (($location = Greece \wedge weather = warm$), $attributes\_clause$, $interest\_score_1$), (($location = Athens \wedge weather = good$), $attributes\_clause$, $interest\_score_2$)}. Both context descriptors in P satisfy the first condition of Def. 12 (i.e., it holds Context($location = Greece \wedge weather = warm$) $covers$ Context($location = Athens \wedge weather = warm$) and ($location = Athens \wedge weather = good$) $covers$ Context($location = Athens \wedge weather = warm$)), but none of them covers the other.

In this case, it is necessary to define which is the most closely related state, i.e., a better match. There are many ways to handle such ties. One is to let the user decide. In this case, both matching preferences are presented to the users, and they decide which one to use. In the next section, we propose two ways of defining similarity among context states.

4.3. State Similarity

We introduce two ways of selecting the most relevant context state. The first one is expressed using the nearest upper level of the hierarchy for each context parameter. The other one selects among the matching context descriptors, the one whose context state has the smallest cardinality. Next, we formalize these two concepts of similarity.

To express similarity between two context states, we introduce a distance function named $hierarchy\ distance$. Using the hierarchy distance leads to choosing the preference that refers to the most specific context state, that is the state that is defined in the most detailed hierarchy level. To define the hierarchy distance, we define first the level of a state as follows.

Definition 13 (Levels of a state) *Given a state $s = (c_1, c_2, \ldots, c_n)$, the hierarchy levels that correspond to this state are $levels(s) = [L_{j_1}, L_{j_2}, \ldots, L_{j_n}]$, such that, $c_i \in dom_{L_{j_i}}(C_i)$, $i = 1, \ldots, n$.*

The distance between two levels is defined as the minimum path between them in a hierarchy, if such a path exists. Otherwise, the distance is infinite.

Definition 14 (Level distance) *Given two levels L_1 and L_2, their distance $dist_H(L_1, L_2)$ is defined as follows:*

1. *if a path exists in a hierarchy between L_1 and L_2, then $dist_H(L_1, L_2)$ is the minimum number of edges that connect L_1 and L_2;*
2. *otherwise $dist_H(L_1, L_2) = \infty$.*

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

Definition 15 (Hierarchy state distance) *Given two states $s^1 = (c_1^1, c_2^1, \ldots, c_n^1)$ and $s^2 = (c_1^2, c_2^2, \ldots, c_n^2)$, the hierarchy distance $dist_H(s^1, s^2)$ is defined as:*

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

A second way to count the distance between two states is to use the Jaccard distance function. In this case, we compute all the descendants of each value of a state. For two values of two states corresponding to the same context parameter, we measure the fraction of the intersection of their corresponding lowest level value sets over the union of this two sets. In this case, we consider as a better match, the "smallest" state in terms of cardinality.

The Jaccard distance of two values v_1 and v_2, belonging to the levels L_i and L_j of the same hierarchy H that has as lowest level the level L_1 can be defined by computing the descendants at the the level L_1, that is the $desc_{L1}^{L_i}(v_1)$ and $desc_{L_1}^{L_i}(v_2)$ of these two values respectively.

Definition 16 (Jaccard distance) *The Jaccard distance of two values v_1 and v_2, belonging to levels L_i and L_j of the same hierarchy H that has as lowest level the level L_1, is defined as:*

$$dist_J(v_1, v_2) = 1 - \frac{desc_{L_1}^{L_i}(v_1) \bigcap desc_{L_1}^{L_j}(v_2)}{desc_{L_1}^{L_i}(v_1) \bigcup desc_{L_1}^{L_j}(v_2)}.$$

Definition 17 (Jaccard state distance) *Given two states $s^1 = (c_1^1, c_2^1, \ldots, c_n^1)$ and $s^2 = (c_1^2, c_2^2, \ldots, c_n^2)$, the Jaccard distance $dist_H(s^1, s^2)$ is defined as*

$$dist_J(s^1, s^2) = \sum_{i=1}^{n} |dist_J(c_i^1, c_i^2)|.$$

We shall show that the ordering of states produced by the Jaccard distance is consistent with the ordering produced by the hierarchy distance.

Property 1 *Assume three values, v_1, v_2, v_3, defined at different levels $L_1 \prec L_2 \prec L_3$ of the same hierarchy having L_0 as the most detailed level, such that $v_3 = anc_{L_2}^{L_3}(v_2) = anc_{L_1}^{L_2}(v_1)$. Then, $dist_J(v_3, v_1) \geq dist_J(v_2, v_1)$.*

Proof: By definition,

$$dist_J(v_1, v_2) = 1 - \frac{desc_{L_0}^{L_1}(v_1) \bigcap desc_{L_0}^{L_2}(v_2)}{desc_{L_0}^{L_1}(v_1) \bigcup desc_{L_0}^{L_2}(v_2)} \ (1)$$

and

$$dist_J(v_1, v_3) = 1 - \frac{desc_{L_0}^{L_1}(v_1) \bigcap desc_{L_0}^{L_3}(v_3)}{desc_{L_0}^{L_1}(v_1) \bigcup desc_{L_0}^{L_3}(v_3)}. \ (2)$$

In both fractions, the numerator reduces to $desc_{L_0}^{L_1}(v_1)$, clearly due to the transitivity property of the ancestor functions. The denominator of the first fraction is $desc_{L_0}^{L_2}(v_2)$, whereas the denominator of the second fraction is $desc_{L_0}^{L_3}(v_3) \supseteq desc_{L_0}^{L_2}(v_2)$, again due to the transitivity property of the ancestor function (i.e, all descendants of v_2 at the detailed level are also descendants of v_3). Therefore $dist(v_3, v_1) \geq dist(v_2, v_1)$.

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

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

Proof: Let $level(s^1)$ = $[L_1^1, L_2^1, \ldots, L_n^1]$, $level(s^2)$ = $[L_1^2, L_2^2, \ldots, L_n^2]$ and $level(s^3)$ = $[L_1^3, L_2^3, \ldots, L_n^3]$. From Def. 10, since s^2 covers s^1, and the fact that the level of any ancestor of c_i is larger than the level of c_i, it holds $L_i^2 \succeq L_i^1$, $\forall\ 1 \leq i \leq n$ (1). Similarly, since s^3 covers s^1, it holds $L_i^3 \succeq L_i^1$, $\forall\ 1 \leq i \leq n$ (2), and, since s^3 covers s^2, it holds $L_i^3 \succeq L_i^2$, $\forall\ 1 \leq i \leq n$ (3). From (1), (2) and (3), we get $L_i^3 \succeq L_i^2 \succeq L_i^1$, $\forall\ 1 \leq i \leq n$ (4). Since $s^2 \neq s^3$, for at least one j, $1 \leq j \leq n$, it holds $L_j^3 \succ L_j^2$ (5). Thus from (4), (5) and Def. 15, it holds $dist_H(s^3, s^1) > dist_H(s^2, s^1)$.

The property states that between two covering states s^2 and s^3, the matching one is the one with the smallest hierarchy distance. Due to Property 1, the same holds for the Jaccard distance as well. That is:

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

4.4. A Context Resolution Algorithm

As already mentioned, given a database with information and a certain context descriptor (that characterizes either the current or a hypothetical context), the problem is to locate the tuples of the relation that correspond to the given context descriptor and score them appropriately.

The problem is further divided in two parts:

1. Locate in the profile tree the paths (i.e., context states) that correspond to the given context descriptor (in an exact or approximate fashion).

2. On the basis of the leaves of these paths (i.e., expressions of the form $A_i = value, score$), determine the corresponding tuples in the underlying database and annotate them with the appropriate score.

In the following, we detail each of these steps.

Determination of relevant paths in the profile tree. Given a contextual query CQ with an extended context descriptor, for each context state $s = (c_1, c_2, \ldots, c_n)$ in the context of the descriptor, we search the contextual preferences in the profile to locate a state that matches it. To this end, we use the profile tree. If there is a state that exactly matches it, that is a state $(c_1, c_2, \ldots, c_n)$ then the associated preference is returned to the user. Note, that this state is easily located, by a single depth-first-search traversal of the Profile tree. Starting from the root of the tree (level 1), at each level i, we follow the pointer associated with $key = c_i$.

If such a state does not exist, we search for a state s' that matches s. If more than one such state exists, we select the one with the smallest distance, using either the hierarchy or the Jaccard distances.

We use the following $Search\_CS$ algorithm to find a context state in the profile tree that is the most similar with a searched state $s = (c_1, c_2, \ldots, c_n)$. The algorithm descends the profile tree starting from the root node in a breadth first manner. To find the path that covers the searched one, we collect a set of candidate paths, each annotated properly with its distance from the given context state. Algorithm 1 presents the $Search\_CS$ algorithm.

Algorithm 1 Search_CS Algorithm

Input: A node R_P of the *Profile tree*, the searching context state $(c_1, c_2, \ldots, c_n)$, the current distance of each candidate path.
Output: A $ResultSet$ of tuples of the form (Attribute name = attribute value, interest score, distance) characterizing a candidate path whose context state is either the same or best covers the searching context state.

Begin
if $\exists\, x \in R_P$ such that $x = c_i$ **then**
  $Search\_CS(x \rightarrow child, \{c_{i+1}, \ldots, c_n\}, distance)$
else if $\forall\, y \in R_P$ such that $y = anc^{L_j}_{L_i}(c_i)$ **then**
  $Search\_CS(y \rightarrow child, \{c_{i+1}, \ldots, c_n\}, dist(y, c_i) + distance)$
else if R_P is a leaf of the form $(A_i = value, score)$ **then**
  $ResultSet = ResultSet \bigcup (A_i = value, score, distance)$
end if
End

Given a *Profile tree* whose root node is R_P, the algorithm returns all paths whose context state is either the same or covers the searching context state $(c_1, c_2, \ldots, c_n)$. Each candidate path counts the distance from the searching path. To search an extended context state, at first we invoke $Search\_CS(R_P, \{c_1, c_2, \ldots, c_n\}, 0)$. At the end of the execution of this call, we can sort all the results on the basis of their distance and select the one with the minimum distance, i.e., the one that differs the least from the searched path based on one of the distances. Clearly the last step can be easily replaced by a simple runtime check that keeps the current closest leaf if its distance is smaller than the one currently tested. Still, we prefer to keep this variant of the algorithm to cover the general case where more than one candidates can be selected by the system or the user.

We show that the algorithm is correct, i.e., if applied for all extended context states specified by the extended context descriptor of the query, it leads to the desired set of states according to Def. 12. For each state, the algorithm returns a state that is the most similar with the searching one, that is the one that has the smallest distance. By Property 2 for the hierarchy distance and Property 3 for the Jaccard distance, it is clear that the state with the smallest distance is one that *covers* the searching state. The set of extended context states that are returned, specify an extended context descriptor. This descriptor covers the query's descriptor, because each state is expressed by another similar one. Furthermore, the textually described variant can give the "best" matching descriptor because for each state we select the "best" matching state.

Query Complexity. To compute the time complexity of a query using the profile tree, we study two cases. In the first one, we are interested in finding only those preferences that their context states exact match with the context state of a query. Here, a query is a simple traversal on the profile tree from the root to leaves. At level i, we search for the cell having as key the i^{th} value in the query and descend to the next level, following the pointer of the corresponding cell. For a profile tree with n context parameters $(C_1, C_2, \ldots, C_n)$, if each parameter has $|edom(C_i)|$ values in its domain, the maximum number of cells that are required to be visited for a query is $|edom(C_1)| + |edom(C_2)| + \ldots + |edom(C_n)|$. So, each query involves exactly such node visits as the height of the tree minus one, i.e., as the number of the context parameters. Otherwise, if we do not use the profile tree, and we scan sequentially the user preferences, in the worst case it is necessary to visit $|edom(C_1)| \times |edom(C_2)| \times \ldots \times |edom(C_n)|$ cells.

In the second case, we are not interested only in exact matches, but we search to find all context states that cover the query state. As described in the previous algorithm, at level i, we search for the cell having as key the i^{th} value in the query and for all the other cells that have relevant values from the upper levels of the i^{th} value in the query. We descend to the next level of the profile tree, following all the corresponding pointers. So, for a profile tree with n context parameters $(C_1, C_2, \ldots, C_n)$, suppose that each parameter has $|edom(C_i)|$ values in its domain and each parameter has h_i hierarchy levels. The maximum number of cells that are required to be visited for a query is $|edom(C_1)| + |edom(C_2)| \times h_1 + |edom(C_3)| \times h_2 \times h_1 + \ldots + |edom(C_n)| \times h_{n-1} \times \ldots \times h_1$. As before, the worst case for a sequential scan, in order to find the appropriate to the query user preferences, needs to visit $|edom(C_1)| \times |edom(C_2)| \times \ldots \times |edom(C_n)|$ cells.

Determination of the database tuples that corre-

spond to the identified states. Assume a relation $R(A_1, A_2, ...A_n)$ and a profile tree P with leaves containing expressions of the form $(A_i = value, score)$. The problem now is that given a context descriptor *cod*, we need to rank the tuples of relation R with respect to *cod*. A simple algorithm is employed for this task.

Algorithm 2 Rank_CS Algorithm

Input: A profile tree P, a relation $R(A_1, A_2, ...A_n)$ and a context descriptor *cod*
Output: A $TupleResultSet$ of tuples of R ranked by the appropriate score.
Variables: A (initially empty) $ExprResultSet$ of expressions of $Search\_CS$ results.

Begin
$\forall$ state $s \in context(cod)$ {
Pick minimum distance tuple e from the result of $Search\_CS(P, s, 0)$
$ExprResultSet = ExprResultSet \bigcup e$
}
$\forall$ expressions e : $(A_i = value, score) \in ExprResultSet$ {
$ResultSet = ResultSet \bigcup \sigma_{A_i=value}(R)$, with the latter annotated with score.
}
End

The algorithm $Search\_CS$ is invoked for all extended context states specified by the query's descriptor. Each invocation returns an expression that characterizes one or more tuples of the underlying relation. Then, we perform all the produced expressions as selections of the relational algebra over the underlying relation. It is straightforward (and practically orthogonal to our problem) to add (a) ranking of the expressions by their score (and consequently, ranking of the results of the queries over the relation) and (b) removal of duplicate tuples produced by these selection queries by keeping the max (equivalently, avg, min, or some wieghted average) for the score of tuples appearing more than once in the $ResultSet$.

5. Evaluation

We evaluate our approach along two perspectives: usability and performance.

5.1. Usability Evaluation

We use a real database of points-of-interest of the two largest cities in Greece, namely Athens and Thessaloniki. To ease the specification of contextual preferences, we created a number of default profiles based on the (a) age (below 30, between 30-50, above 50), (b) sex (male or female) and (c) taste (broadly categorized as mainstream or out-of-the-beaten track). Based on the above three characteristics, users were assigned one of the 12 available profiles. Users were allowed to modify the default profiles assigned to them by adding, deleting or updating preferences. We evaluated the system along two lines: easy of profile specification and quality of results. We run our prototype implementation for 10 users; the results are summarized in Table 1. For all users, it was the first time that they used the system.

With regards to profile specification, we count the number of modifications (insertions, deletions, updates) of preferences of the default profile that was originally assigned to the users. In addition, we report how long (in minutes), it takes users to specify/modify their profile. This also includes the time it took users to understand how profile specification works. The results are reported in the first two lines of Table 1. The general impression was that predefined profiles save time in specifying user preferences. Furthermore, having default profiles makes it easier for someone to understand the main idea behind the system, since the preferences in the profile act as examples. With regards to time, as expected, there is deviation among the time users spent on specifying profiles: some users were more meticulous than others, spending more time in adjusting the profiles assigned to them.

With regards to the quality of the results, users were asked to rank the results of each contextual query manually. Then, we compare the ranking specified by the users with what was recommended by the system, for the following three cases: (i) when there is an exact match (ii) when there is exactly one cover, and (iii) when there is more than one cover, and the $Hierarchy$ or the $Jaccard$ distance functions are used. For each case, we consider the best 20 results, i.e., the 20 points-of-interest that were ranked higher. When there are ties in the ranking, we consider all results with the same score. We report the percentage of the results returned that belong to the results given by the user. As shown in Table 1, this percentage is generally high. Surprisingly, sometimes users do not conform even to their own preferences as shown by the results for exact match queries. In this case, although the context state of the preferences used was an exact match of the context state in the query, still some users ranked their results differently than the system. In such cases, traceability helps a lot, since users can track back which preferences were used to attain the results and either modify the preferences or reconsider their ranking. Note that users that customized their profile by making more modification got more satisfactory results than those that spend less time during profile specification. Finally, it seems that the $Jaccard$ distance produces more accurate results than the $Hierarchy$ distance mainly because the $Hierarchy$ distance produces rankings with many ties.

Table 1. User Study Results

	User 1	User 2	User 3	User 4	User 5	User 6	User 7	User 8	User 9	User 10
Num of updates	22	31	12	28	24	32	38	13	18	25
Update time (mins)	30	45	20	30	30	40	45	15	20	25
Exact match	100%	90%	90%	95%	90%	100%	100%	85%	100%	100%
1 cover state	100%	95%	90%	85%	90%	100%	100%	85%	90%	100%
More cover states										
Hierarchy	90%	85%	80%	80%	90%	90%	90%	70%	85%	85%
Jaccard	95%	90%	85%	100%	95%	90%	100%	75%	85%	95%

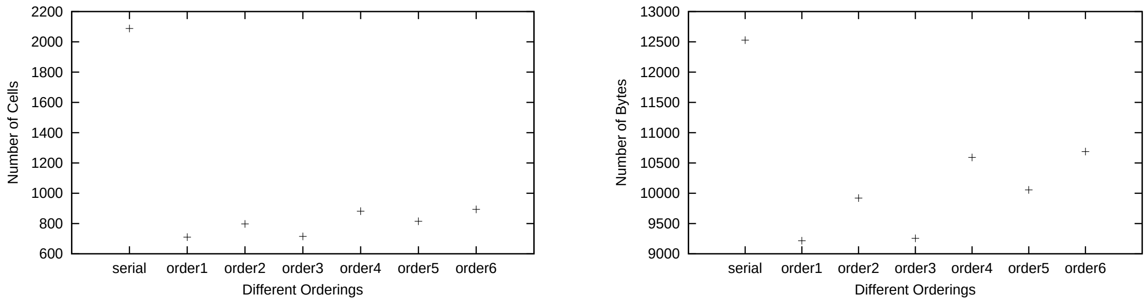

Figure 5. The size of the profile tree using real profiles.

5.2. Performance Evaluation

To evaluate performance, we run a set of experiments using both real and synthetic profiles. The *real profile* is the one used for the usability study. We consider: (a) the space required to store preferences when using a profile tree as opposed to storing them sequentially and (b) the complexity of context resolution.

Size of the Profile Tree In this set of experiments, we evaluate the size of the context tree for different mappings of the context parameters to the levels of the tree.

Using a Real Profile. The real profile has 522 user preferences. Each preference consists of three context values ($accompanying\_people$, $time$, $location$), an $attribute\_name$, an $attribute\_value$ and an *interest score*. The active domains of the context parameters have 4, 17, 100 values, respectively. We count the total number of cells and the total number of bytes of the context tree that is created for each ordering of the context parameters. Let A stand for $accompanying\_people$, T for $time$ and L for $location$. We call $order$ 1 the ordering in which A is assigned to the first level of the tree, T to the second and L to the third one, that is the ordering *(A, T, L)*. $Order$ 2 is the ordering *(A, L, T)*, $order$ 3 is *(T, A, L)*, $order$ 4 is *(T, L, A)*, $order$ 5 is *(L, A, T)* and $order$ 6 is *(L, T, A)*. As shown in Fig. 5, the orderings that result in trees with smaller sizes are the ones that map the context parameter with large domains lower in the tree. In addition, all trees occupy less space than storing preferences sequentially.

Using Synthetic Profiles. We study the size of the tree as a function of the size of the profile (i.e, number of user preferences). Synthetic profiles have three context parameters, and thus, the profile tree has three levels (plus one for the leaves). There are three different types regarding the cardinality of the domains of the context parameters: a domain with 50 values, a domain with 100 values and a domain with 1000 values and profiles with various numbers (500, 1000, 5000 and 10000) of user preferences. Context values are drawn from their corresponding domain, either using a uniform data distribution, or a zipf data distribution with a = 1.5. The size of the tree depends on the ordering of context parameters. We call $order$ 1 the ordering in which the parameter whose domain has 50 values is assigned to the first level, the parameter with 100 values to the second one, and the parameter with 1000 values to the last one. $Order$ 2 is the ordering $(50, 1000, 100)$, $order$ 3 is $(100, 50, 1000)$, $order$ 4 is $(100, 1000, 500)$, $order$ 5 is $(1000, 50, 100)$ and $order$ 6 is $(1000, 100, 50)$. As expected, storage is minimized when the parameters with large domains are placed lower in the tree (Fig. 6 (left, center)). For the zipf distribution (Fig. 6, center), the total number of cells is smaller than for the uniform distribution (Fig. 6, left), because using the zipf distribution “hot” values appear more frequently in preferences, i.e., more context values are the same.

In the last experiment, we show that the best way of assigning parameters to levels depends on the percentage of values from the domain of each parameter that $actually$ appears in the preferences. Thus, if a parameter has a very skewed data distribution, it may be more space efficient to map it higher in the tree, even if its domain is large (Fig. 6, right). In this experiment, the profile has 5000 preferences, and the context parameters have domains with 50, 100 and 200 values. The values of the parameters with domains with 50 and 100 values are selected using a $uniform$ data distribution and the values of the parameter with 200 values using a $zipf$ data distribution with various values for the parameter a, varying from 0 (corresponding to the $uniform$ distribution) to 3.5 (corresponding to a very high skew). $Order$ 1 is the ordering $(50, 100, 200)$, $order$ 2 is $(50, 200, 100)$ and $order$ 3 is $(200, 50, 100)$.

Number of Cells Accesses To study the usefulness of the profile tree in answering preference queries, and in particular for finding the appropriate preferences, we performed a set of experiments in which we count the number of cell accesses during context resolution. We run this experiment using both the real (Fig. 7, left) and synthetic profiles (Fig. 7, center and right). We use synthetic profiles with 500, 1000, 5000, and 10000 preferences. In all cases, the profile tree is the one produced when the larger domains are mapped in levels lower in the tree. In synthetic profiles, the context values are selected from the corresponding domain, either using a uniform data distribution, or zipf data distribution with a = 1.5. We run this experiment for 50 different queries, where the context parameters have values from different hierarchy levels. The parameter with 50 values has 2 hierarchy levels, the parameter with 100 values has 3 hierarchy levels, and the parameter with 1000 values has 3 hierarchy levels. With the profile tree, exact match queries are resolved by a simple root-to-leaf traversal, while non exact matches need to consider multiple candidate paths. In the case of the sequential scan, for exact matches, the profile is scanned until the matching state is found, while for non exact matches, we need to scan the whole profile.

6. Related Work

Although there has been a lot of work on developing a variety of context infrastructures and context-aware middleware and applications (such as, the Context Toolkit [14] and the Dartmouth Solar System [3]), there has been only little work on the integration of context information into databases. Next, we discuss work related to context-aware queries and preference queries. In our previous research ([16, 17, 18]), we have addressed the same problem of expressing contextual preferences. However, the model used there for defining preferences includes only a *single* context parameter. Interest scores of preferences involving more than one context parameter are computed by a simple weighted sum of the preferences of single context parameters. Here, we allow contextual preferences that involve more than one context parameter as well as we associate context with queries. The problem of context state resolutions and its development is also new in this paper.

In [10], 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. Situations are uniquely linked with preferences through a N:M relationship. A concrete situated preference expresses that a preference holds in a specific situation.

Context and Queries Although, there is much research on location-aware query processing in the area of spatio-temporal databases, integrating other forms of context in query processing is less explored. In the context-aware querying processing framework of [8], there is no notion of preferences, instead context attributes are treated as normal attributes of relations. Storing context data using data cubes, called context cubes, is proposed in [9] for developing context-aware applications that use archive sensor data. In this work, data cubes are used to store historical context data and to extract interesting knowledge from large collections of context data. The Context Relational Model (CR) introduced in [13] is an extended relational model that allows attributes to exist under some contexts or to have different values under different contexts. CR treats context as a first-class citizen at the level of data models, whereas in our approach, we use the traditional relational model to capture context as well as context-dependent preferences. Context as a set of dimensions (e.g., context parameters) is also considered in [15] where the problem of representing context-dependent semistructured data is studied. A similar context model is also deployed in [7] for enhancing web service discovery with contextual parameters. Recently, context has been used in information filtering to define context-aware filters which are filters that have attributes whose values change frequently [6]. Finally, in [12], the current contextual state of a system is represented as a multidimensional subspace within or near other situation subspaces.

Preferences in Databases In this paper, we use context to confine database querying by selecting as results the best matching tuples based on the user 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. With the *quantitative approach*, prefer-

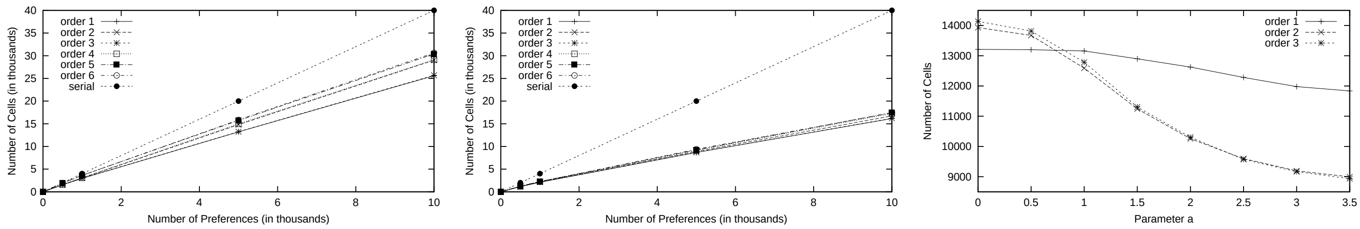

Figure 6. Uniform (left), zipf with a=1.5 (center) and combined (right) data distribution.

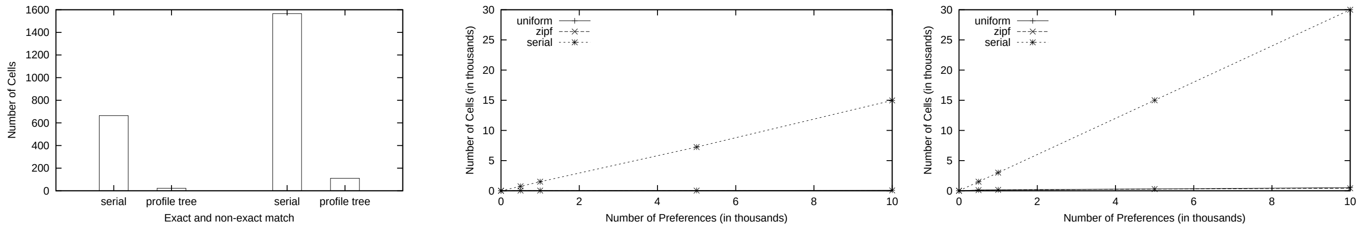

Figure 7. Number of cells accessed to find related preferences to queries, in real profiles (left) and in synthetic profiles, for an exact (center) and non exact match (right).

ences are expressed indirectly by using scoring functions that associate a numeric score with every tuple of the query answer. In our work, we have adapted the general quantitative framework of [1], since it is more easy for users to employ. In the quantitative framework of [11], user preferences are stored as degrees of interest in *atomic query elements* (such as individual selection or join conditions) instead of interests in specific attribute values. 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. In the *qualitative approach* (for example, [4]), the preferences between the tuples in the answer to a query are specified directly, typically using binary preference relations. This framework can also be readily extended to include context.

7. Summary

In this paper, we focus on handling contextual preferences. We define context descriptors for specifying conditions on context parameters that allow the specification of context states at various levels of detail. Preferences are augmented with context descriptors that specify their scope of applicability. Similarly, queries are enhanced with context descriptors that define the context of their execution. We formulate the problem of identifying the preferences that are most relevant to the context of a query. To address this problem, we develop the notion of cover between states as well as appropriate distance functions. We also present an algorithm that locates the relevant preferences. We also introduce index structures that exploit contextual information for (a) storing preferences and (b) caching the results of queries based on their context.

References

[1] R. Agrawal and E. L. Wimmers. A framework for expressing and combining preferences. In *ACM SIGMOD COMAD*, pages 297–306, 2000.

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

[3] G. Chen, M. Li, and D. Kotz. Design and implementation of a largescale context fusion network. In *MobiQuitous*, 2004.

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

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

[6] J.-P. Dittrich, P. M. Fischer, and D. Kossmann. Agile: Adaptive indexing for context-aware information filters. In *ACM SIGMOD COMAD*, pages 215–226, 2005.

[7] C. Doulkeridis and M. Vazirgiannis. Querying and Updating a Context-Aware Service Directory in Mobile Environments. *Web Intelligence*, pages 562–565, 2004.

[8] L. Feng, P. Apers, and W. Jonker. Towards Context-Aware Data Management for Ambient Intelligence. In *DEXA*, 2004.

[9] L. D. Harvel, L. Liu, G. D. Abowd, Y.-X. Lim, C. Scheibe, and C. Chatham. Context cube: Flexible and effective manipulation of sensed context data. In *Pervasive*, pages 51–68, 2004.

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

[11] G. Koutrika and Y. Ioannidis. Personalization of queries in database systems. *ICDE*, 00:597, 2004.

[12] A. Padovitz, S. W. Loke, and A. Zaslavsky. Towards a theory of context spaces. *PerCom*, 00:38, 2004.

[13] Y. Roussos, Y. Stavrakas, and V. Pavlaki. Towards a Context-Aware Relational Model. *CRR*, 2005.

[14] D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *CHI*, pages 434–441, 1999.

[15] Y. Stavrakas and M. Gergatsoulis. Multidimensional Semistructured Data: Representing Context-Dependent Information on the Web. *CAiSE*, 2002.

[16] K. Stefanidis, E. Pitoura, and P. Vassiliadis. On Supporting Context-Aware Preferences in Relational Database Systems. *MCMP*, 2005.

[17] K. Stefanidis, E. Pitoura, and P. Vassiliadis. A Context-Aware Preference Database System. *International Journal of Pervasive Computing and Communications*, 2006. Accepted, December 2005. To appear.

[18] K. Stefanidis, E. Pitoura, and P. Vassiliadis. Modeling and Storing Context-Aware Preferences. *ADBIS*, 2006. Accepted, June 2006. To appear.

[19] P. Vassiliadis and S. Skiadopoulos. Modelling and Optimisation Issues for Multidimensional Databases. In *CAiSE*, 2000.