Preference Queries in Context

Kostas Stefanidis
Department of Computer Science
University of Ioannina
GR-45110 Ioannina, Greece
kstef@cs.uoi.gr

Evaggelia Pitoura
Department of Computer Science
University of Ioannina
GR-45110 Ioannina, Greece
pitoura@cs.uoi.gr

Panos Vassiliadis
Department of Computer Science
University of Ioannina
GR-45110 Ioannina, Greece
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. In general, users may have different preferences depending on context, for instance, on the current weather, time or location. In this paper, we define a model for expressing contextual preferences. We model context as an ordered set of multidimensional attributes. Then, user preferences can be specified as functions over these context attributes. We formulate the problem of identifying the preferences that are most relevant to a query and present an algorithm that locates them. We also introduce index structures that exploit contextual information for (a) storing preferences and (b) caching the results of queries based on their context.

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, in the sense that, each time, we pick the state which is most specific with respect to the query context. 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 propose storing the resolved

context states as well as the results of context queries in a *context query tree*, so that these results may be used by similar queries in the future. The main difference between the profile and the context query tree is that the profile tree stores the context states expressed by the users in their preference, while the context query tree maintains the context states that appear in users' queries.

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 5, 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 $\{C_1, C_2, \ldots, C_n\}$. 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 $(c_1, c_2, \ldots, c_n)$, 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: $W = dom(C_1) \times dom(C_2) \times \cdots \times dom(C_n)$.

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 $(L, \prec)$: $L = (L_1, \ldots, L_{m-1}, ALL)$ 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 (left). Levels of *location* are *Region*, *City*, *Country*, and *ALL*. *Region* is the most detailed level. Level *ALL* is the most coarse level.

The relationship between the values of the context levels is achieved through the use of the set of $anc^{L_j}_{L_i}$, $L_i \prec L_j$, functions [20]. A function $anc^{L_j}_{L_i}$ assigns a value of the domain of L_i to a value of the domain of L_j. For instance, $anc^{City}_{Region}(Plaka) = Athens$. Formally, the set of functions $anc^{L_j}_{L_i}$ satisfies the following conditions:

1. For each pair of levels L_1 and L_2 such that $L_1 \prec L_2$, the function $anc^{L_2}_{L_1}$ 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 $anc^{L_3}_{L_1}$ equals to the composition $anc^{L_1}_{L_2} \circ anc^{L_2}_{L_3}$.
3. For each pair of levels L_1 and L_2 such that $L_1 \prec L_2$, the function $anc^{L_2}_{L_1}$ is monotone, i.e., $\forall\ x, y \in dom_{L_1}(C_i)$,

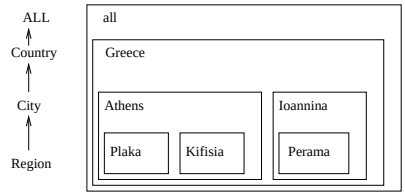

Figure 1: Hierarchies on *location*.

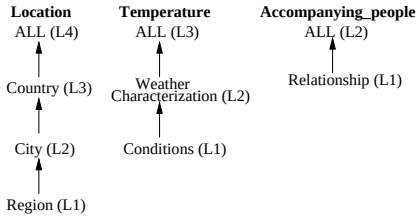

Figure 2: Hierarchies on *location*, *temperature* and *accompanying_people*.

$$L_1 \prec L_2,\, x < y \Rightarrow anc_{L_1}^{L_2}(x) \leq anc_{L_1}^{L_2}(y).$$

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 (right) 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 $(c_1, c_2, \ldots, c_n)$, 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: $EW = edom(C_1) \times edom(C_2) \times \cdots \times edom(C_n)$.

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:*

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

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 $cod = cod(C_{i_1}) \wedge cod(C_{i_2}) \wedge \cdots \wedge cod(C_{i_k})$, $0 \leq k \leq n$. 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. The context states that belong to more than one conflicting preference are identified and they are not maintained by the system. Finally, we define *profile* P as:

DEFINITION 7 (PROFILE). *A profile P is a set of non-conflicting contextual preferences.*

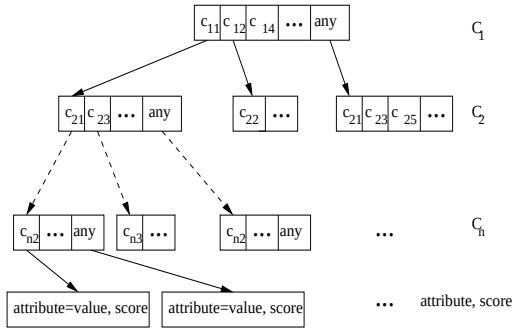

Figure 3: The profile tree.

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.

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 there is one additional level for the leaves. Thus, the total 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.

the context parameters are assigned to the levels of the contextual profile tree importance. So, as smaller is the weight of a parameter, as lower is placed in

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 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: $contextual\_preference_1 = ((\text{location} = Kifisia \wedge \text{temperature} = warm \wedge \text{accompanying\_people} = friends), (\text{type} = cafeteria), 0.9)$, $contextual\_preference_2 = ((\text{accompanying\_people} = friends), (\text{type} = brewery), 0.9)$, and $contextual\_preference_3 = ((\text{location} = Plaka \wedge \text{temperature} \in \{warm, hot\}), (\text{name} = Acropolis), 0.8)$. 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. Leaf nodes store the relative to each contextual preference attribute name with its value, and the interest score of the corresponding preference. 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 $m_1 \leq m_2 \leq \cdots \leq m_n$, 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

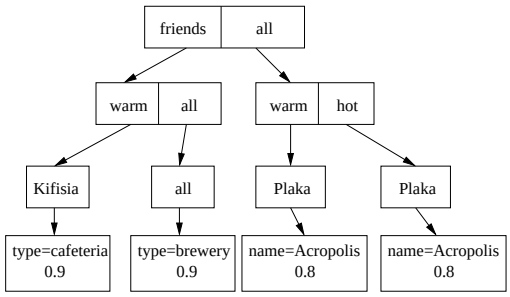

Figure 4: An instance of a profile tree.

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:* $(cod_{11} \wedge \cdots \wedge cod_{1j}) \vee \cdots \vee (cod_{l1} \wedge \cdots \wedge cod_{lm})$ *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* $s^1 = (c_1^1, c_2^1, \ldots, c_n^1) \in EW$ *covers an extended context state* $s^2 = (c_1^2, c_2^2, \ldots, c_n^2) \in EW$, *iff* $\forall\ k,\ 1 \leq k \leq n,\ c_k^1 = c_k^2$, *or* $c_k^1 = anc_{L_i}^{L_j}(c_k^2)$ *for some levels* $L_i \prec L_j$.

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, $c_k^1 = anc_{L_2}^{L_1}(c_k^2)$ and $c_k^2 = anc_{L_1}^{L_2}(c_k^1)$. 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 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,

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

and

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

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 $y = anc_{L_i}^{L_j}(c_i)$ **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.

Determination of the database tuples that correspond 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. CACHING CONTEXTUAL QUERIES

In this section, we present a scheme for storing (caching) results of previous queries executed at a specific context, so that these results can be re-used by subsequent queries.

5.1 The Contextual Query Tree

To store the results of previous queries, we use a data structure similar to the profile tree, called *contextual query tree* (Fig. 5). Similarly with the profile tree, a contextual query tree has $n + 1$ levels. Each of the n context parameters is assigned to the first n levels of the tree and there is an additional level for the leaves. For simplicity, assume that context parameter C_i is mapped to level i. A path in the context tree corresponds to an *extended context state*, i.e., an assignment of values to context parameters. In the leaves of the tree, we store (cache) the results of previous queries, so that they can be re-used by subsequent ones.

The contextual query tree is constructed incrementally each

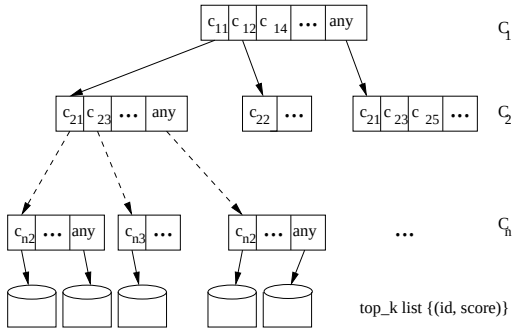

Figure 5: The contextual query tree.

time a contextual query is computed. Each non-leaf node at level k contains cells of the form $[key, pointer]$, where key is equal to $c_{kj} \in edom(C_k)$ for a value of the context parameter C_k that appeared in some previously computed 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 of a contextual 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.

Besides just storing the context states that appear in the query, in the case in which there is no exact matching state in a contextual preference, we also store the best matching state as produced by the context resolution algorithm (Algorithm 1). For instance, given a state $(Athens, warm, any)$ specified by the context descriptor $(location = Athens \wedge weather = warm)$, if the state does not exist in any contextual preference and its matching state returned by Algorithm 1 is $(Europe, warm, any)$, we store both states in the tree and link them together. This allows us to save the cost of resolution as well as to re-use any results for both states.

An issue is what is stored in the leaves. In our running example, we may store a list of ids, e.g., $points\_of\_interest$ ids, along with their interest scores for the associated context state, that is, for the path from the root leading to them. Instead of storing scores for all non-context parameters, to be storage-efficient, we may just store the $top-k$ ids (keys), that is the ids of the items having the k-highest 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 of their scores.

Alternatively, similarly with the profile tree, we may store in the leaves just the associated preference (attribute-clause and interest score). In the case in which the path leading to the leaf does not correspond exactly to a path at the profile tree, we store as well its distance from the actual state.

The way that the context parameters are assigned to the levels of the contextual query tree affects its size. If the domain of the first level of the tree, i.e., the root of the tree, has n_0 values (including the any value), the second level n_1

values, and the last one n_k, then the maximum number of cells is $n_0 * (1 + n_1 * (1 + \ldots (1 + n_k)))$. The above number is as small as possible, when $n_0 \leq n_1 \leq \cdots \leq n_k$, thus, it is better to place context parameters with domains with higher cardinalities lower in the context tree.

Finally, there are two additional issues related to managing the contextual query tree: replacement and update. To bound the space occupied by the tree, standard cache replacement policies, such as LRU or LFU, may be employed to replace the entry, that is the path, in the tree that was least frequently or least recently used. Regarding cache updates, stored results may become obsolete, either because there is an update in the contextual preferences or because entries (points_of_interests, in our running example) are deleted, inserted or updated. In the case of a change in the contextual preferences, we update the contextual query tree by deleting the entries that are associated with paths, that is context states, that are involved in the update. In the case of updates in the database instance, we do not update the tree, since this would induce high maintenance costs. Consequently, some of the scores of the entries cached in the tree may be invalid. Again, standard techniques, such periodic cache refreshment or associating a time-out with each cache entry, may be used to control the deviation between the cached and the actual scores.

5.2 Querying the Contextual Query Tree

Using the contextual query tree, we speed up query processing for a submitted query. Assume a contextual query with an extended context descriptor that specifies a set of extended context states. Before computing the relative to each state results (using the profile tree), we check the contextual query tree to see if some of the states exist. If so, we avoid to compute the *top-k* results for these states. Else, we compute the *top-k* results, using the profile to find the appropriate preferences, and insert the new context state, i.e., the new path, and the associated *top-k* results in the tree.

Note that by using the tree, we also save the cost of context resolution for previously resolved queries. Assume that we have stored state s^1 and the fact that state s^2 is its best match. When a query with state s^1 or s^2 is submitted, we can re-use the stored results for both states s^1 and s^2. Further, we save the cost for all states s, such that s *covers* s^1 and s^2 *covers* s. By just traversing the contextual query tree, we can deduce that s does not appear in the profile tree and that its best match is s^2. This can be easily proved by contradiction as follows. Assume that s appears in the profile tree. Then, since s *covers* s^1 and s^2 *covers* s, we should have selected s as a best match for s^1 instead of s^2, which is not the case. Thus, s does not appear in the profile tree. Also, s^2 is its best match, since if another s' was its best match, then s' should be the best match for s^1 as well, which is again a contradiction.

The search process for a context state is a simple traversal of the contextual query tree. At level i, we search in a node for the cell having as key value the i^{th} value of the searched context state and descend to the next lower level, following the appropriate pointer. For a context 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 $|edom(C_1)| + |edom(C_2)| + \cdots + |edom(C_n)|$. So, each search in the contextual query tree is fast, simply, because it requires exactly as many node visits as the height of the tree minus one, i.e., as the number of the context parameters.

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 [15] 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 ([18, 19], 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.

6.1 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 [14] 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 [17] 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, context has been used in the area of multidatabase systems to resolve semantic differences, e.g., [10, 13, 16] and as a general mechanism for partitioning information bases [12].

6.2 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*, preferences 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.

Acknowledgment

This research was funded by the program "Pythagoras" of the Operational Program for Education and Initial Vocational Training of the Hellenic Ministry of Education under the 3^{rd} Community Support Framework and the European Social Fund.

8. REFERENCES

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

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

[3] 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.

[4] J. Chomicki. Preference Formulas in Relational Queries. *TODS*, 28(4), Dec 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 *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 215–226, New York, NY, USA, 2005. ACM Press.

[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 *Proc. of the 15th Intl. Conf. on Database and Expert Systems Applications (DEXA)*, 2004.

[9] 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.

[10] V. Kashyap and A. Sheth. So Far (Schematically) yet So Near (Semantically). In *Proc. of IFIP WG 2.6 Database Semantics Conference on Interoperable Database Systems (DS-5)*, 1992.

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

[12] J. Mylopoulos and R. Motschnig-Pitrik. Partitioning Information Bases with Contexts. In *Proc. of CoopIS*, 1995.

[13] A. Ouksel and C. Naiman. Coordinating Context Building in Heterogeneous Information Systems. *J. Intell Inf Systems*, 3, 1993.

[14] 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.

[15] 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.

[16] E. Sciore, M. Siegel, and A. Rosenthal. Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems. *ACM TODS*, 1994.

[17] Y. Stavrakas and M. Gergatsoulis. Multidimensional Semistructured Data: Representing Context-Dependent Information on the Web. *International Conference on Advanced Information Systems Engineering (CAiSE 2002)*, 2002.

[18] K. Stefanidis, E. Pitoura, and P. Vassiliadis. On Supporting Context-Aware Preferences in Relational Database Systems. *International Workshop on Managing Context Information in Mobile and Pervasive Environments*, 2005.

[19] K. Stefanidis, E. Pitoura, and P. Vassiliadis. Modeling and Storing Context-Aware Preferences. Submitted. Available at: http://www.cs.uoi.gr/∼kstef/, 2006.

[20] P. Vassiliadis and S. Skiadopoulos. Modelling and Optimisation Issues for Multidimensional Databases. In *Proc. of 12th International on Advanced Information Systems Engineering,(CAiSE 2000)*, 2000.