Modeling and Storing Context-Aware Preferences

Kostas Stefanidis, Evaggelia Pitoura, and Panos Vassiliadis

Department of Computer Science, University of Ioannina, Greece
{kstef, pitoura, pvassil}@cs.uoi.gr

Abstract. Today, the overwhelming volume of information that is available to an increasingly wider spectrum of users creates the need for personalization. In this paper, we consider a database system that supports context-aware preference queries, that is, preference queries whose result depends on the context at the time of their submission. We use data cubes to store the associations between context-dependent preferences and database relations and OLAP techniques for processing context-aware queries, thus allowing the manipulation of the captured context data at different levels of abstractions. To improve query performance, we use an auxiliary data structure, called context tree, which indexes the results of previously computed preference-aware queries based on their associated context. We show how these cached results can be used to process both exact and approximate context-aware preference queries.

1 Introduction

The increased amount of available information creates the need for personalized information processing [1]. Instead of overwhelming the user with all available data, a personalized query returns only the relevant to the user information. In general, to achieve personalization, users express their preferences on specific pieces of data either explicitly or implicitly. The result 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 situation of an entity [2]. 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* [3]. A *context-aware* system is a system that uses context to provide relevant information and/or services to its users. In this paper, we consider a *context-aware* preference database system that supports preference queries whose results depend on context. In particular, users express their preferences on specific attributes of a relation. Such preferences depend on context, that is, they may have different values depending on context.

We model context as a finite set of special-purpose attributes, called *context parameters*. Users express their preferences on specific database instances based on a single context parameter. Such *basic preferences*, i.e., preferences associating database relations with a single context parameter, are combined to compute *aggregate preferences* that include more than one context parameter. Context parameters may take values for hierarchical domains, thus different levels of abstraction for the captured context data are introduced. For instance, this allows us to represent preference along the location context parameter at different levels of detail, for example, by grouping together preferences for all cities of a specific country. Basic preferences are stored in data cubes, following the OLAP paradigm.

Although, aggregate preferences are not explicitly stored, we cache the results of previously computed preference queries using a data structure called *context tree*. The context tree indexes the results of queries based on their associated context. The cached results are re-used to speed up the processing of queries that refer to the exact context of a previously computed query as well as of queries whose context is similar enough to those of some previously computed ones. We provide initial experimental results that characterize the quality of the approximation attained by using preferences computed at similar context states.

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

– We provide a logical model for context-aware preferences that is based on a multidimensional model of context.
– We propose storing the results of previously computed preference queries using a data structure, the *context tree*, that indexes these results based on the values of the context parameters.
– We show how cached results can be used to compute both exact and approximate context-aware preference queries.

2 A Logical Model for Context and Preferences

2.1 Reference Example

Consider a database schema with information about *points_of_interest* and *users* (Fig. 1). The *points_of_interest* may for example be museums, monuments, archaeological places, zoos. We consider three context parameters as relevant to this application: *location*, *temperature* and *accompanying_people*. Users have preferences about *points_of_interest* that they express by providing a numeric score between 0 and 1. The degree of interest that a user expresses for a *point_of_interest* depends on the values of the context parameters. For example, a user may visit different places depending on the current temperature, for instance, user *Mary* may give *Acropolis* that is an open-air place, a lower score when the weather is *cold* than when the weather is *warm*. We consider temperature to take one of the following values: *freezing*, *cold*, *mild*, *warm*, and *hot*. Furthermore, the location of users may also affect their preferences, for example, a user may prefer to visit places that are nearby her current location. Similarly, the result of a query depends on the *accompanying_people* that might be *friends*, *family*, and *none*. For example, a *zoo* may be a better place to visit than a *brewery* in the context of *family*.

Points_of_Interest(<u>pid</u>, name, type, location, open-air, hours_of_operation, admission_cost)
User(<u>uid</u>, name, phone, address, e-mail)

Fig. 1. The database schema of our reference example.

2.2 Modeling Context

Context is modeled through a finite set of special-purpose attributes, called *context parameters* (C_i). For a given application X, we define its context environment CE_X as a set of n context parameters $\{C_1, C_2, \ldots, C_n\}$. For instance, the context environment of our example is {*location, temperature, accompanying_people*}. As usual, a *domain* is an infinitely countable set of values. A *context state* corresponds to assigning to each context parameter a value from its domain. For instance, a context state may be: $CS(current) = \{Plaka, warm, friends\}$. The result of a context-aware preference query depends on the context state of its execution. It is possible for some context parameters to participate in an associated *hierarchy of levels* of aggregated data, i.e., they can be viewed from different levels of detail. Formally, an *attribute hierarchy* is a lattice of attributes – called *levels* for the purpose of the hierarchy – $L = (L_1, \ldots, L_n, ALL)$. We require that the upper bound of the lattice is always the level ALL, so that we can group all values into the single value '*all*'. The lower bound of the lattice is called the detailed level of the parameter. In our running example, we consider *location* to be such an attribute as shown in Fig. 2 (left). Levels of *location* are *Region*, *City*, *Country*, and *ALL*. *Region* is the most detailed level, while level *ALL* is the most coarse level.

2.3 Contextual Preferences

In this section, we define how context affects the results of a query. Each user expresses her preference for an item in a specific context by providing a numeric score between 0 and 1. This score expresses a degree of interest: value 1 indicates extreme interest, while value 0 indicates no interest. We distinguish preferences into basic (involving a single context parameter) and aggregate ones (involving a combination of context parameters).

Basic Preferences. Each basic preference is described by (a) a value of a context parameter $c_i \in dom(C_i)$, $1 \leq i \leq n$, (b) a set of values of non-context parameters $a_i \in dom(A_i)$, and (c) a degree of interest, i.e., a real number between 0 and 1. So, for a context parameter c_i, we have:

$$preference_{basic_i}(c_i, a_{k+1}, \ldots, a_m) = interest\_score_i.$$

In our reference example, besides the three context parameters (i.e, *location*, *temperature* and *accompanying_people*), the set of non-context parameters are attributes about *points_of_interest* and *users* that are stored in the database. For example, assume user $Mary$ and the point-of-interest *Acropolis*. When $Mary$ is in the *Plaka* area, she likes to visit *Acropolis* and gives it score 0.8. Similarly, she prefers to visit *Acropolis* when the weather is *warm* and gives *Acropolis* score 0.9. Finally, if she is with $friends$, $Mary$ gives *Acropolis* score 0.6. So, the basic preferences for *Acropolis* and $Mary$ are:

$$preference_{basic_1}(Plaka, Acropolis, Mary) = 0.8,$$

$$preference_{basic_2}(warm, Acropolis, Mary) = 0.9,$$
$$preference_{basic_3}(friends, Acropolis, Mary) = 0.6.$$

For context values not appearing explicitly in a basic preference, we consider a default interest score of 0.5.

Aggregate Preferences. Each aggregate preference is derived from a combination of basic ones. An aggregate preference involves (a) a set of of n values x_i, one for each context parameter C_i, where either $x_i = c_i$ for some value $c_i \in dom(C_i)$ or $x_i = *$, which means that the value of the context parameter C_i is irrelevant, i.e., the corresponding context parameter should not affect the aggregate preference, and (b) a set of values of non-context parameters $a_i \in dom(A_i)$, and has a degree of interest:

$$preference(x_1, \dots x_n, a_{k+1}, \dots, a_m) = interest\_score.$$

The interest score of the aggregate preference is a *value function* of the individuals scores of the basic preferences. This value function prescribes how to combine basic preferences to produce an aggregate score. In general, this may be any computable function specified by the user. In this paper, we assume that the interest score of an aggregate preference is simply a weighted sum of the corresponding basic preferences. Users just specify a weight w_i for each context parameter C_i, such that, $\sum_{i=1}^{n} w_i = 1$. For instance, in the previous example, if the weight of *location* is 0.6, the weight of *temperature* is 0.3 and the weight of *accompanying_people* is 0.1, $preference(Plaka, warm, friends, Acropolis, Mary)$ gets score 0.81.

We describe next two approaches for computing the aggregate scores when the value for some parameters in the preference is '*'. The first one assumes a score of 0.5 for those context parameters whose values in the preference is '*'. Then, the *interest_score* for the preference $preference(x_1, \dots x_n, a_{k+1}, \dots, a_m)$ is computed as:

$$interest\_score = \sum_{i=1}^{n} w_i \times y_i$$

where $y_i = preference_{basic_i}(x_i, a_{k+1}, \dots, a_m)$, if $x_i = c_i$ and $y_i = 0.5$, if $x_i = *$.

The other approach includes in the computation only the interest scores of those context parameters whose values are specified in the preference and ignores those specified as irrelevant. In this case, the *interest_score* for the preference $preference(x_1, \dots x_n, a_{k+1}, \dots, a_m)$ is computed as follows. Assume without loss of generality, that for the first k parameters x_i, $1 \leq i \leq k$, it holds $x_i = c_i$, for $c_i \in dom(C_i)$ and for the remaining $n - k$ parameters, x_i, $k < i \leq n$, it holds $x_i = *$. Then,

$$interest\_score = \sum_{i=1}^{k} w'_i \times y_i$$

where $w'_i = \frac{w_i}{\sum_{j=1}^{k} w_j}$, $y_i = preference_{basic_i}(x_i, a_{k+1}, \dots, a_m)$, if $x_i = c_i$ and $y_i = 0.5$, if $x_i = *$.

For instance, $preference(Plaka, *, friends, Acropolis, Mary)$ has score 0.69 when using the first approach and score 0.77, when using the second one.

It is easy to see that the orderings produced by each approach are consistent with each other [4]. That is, both approaches order the tuples (e.g., the *points_of_interest* in our example) the same way, since in both cases their aggregate score depends on the values of the context parameters that are specified, i.e., are not irrelevant. In the following, we assume that the second approach is used.

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

2.4 Preferences for Hierarchical Context Parameters

When the context parameter of a basic preference participates in different levels of a hierarchy, users may express their preference in any level, as well in more than one level. For example, *Mary* can denote that the monument of *Acropolis* has interest score 0.8 when she is at *Plaka* and 0.6 when she is in *Athens*.

For a parameter L, let L_1, L_2,..., L_n, ALL be the different levels of the hierarchy. There is a hierarchy tree, for each combination of non-context parameters. In our reference example, there is a hierarchy tree for each user profile and for a specific *point_of_interest* that represents the interest scores of the user for the *points_of_interest*, according to the location parameter hierarchy. In Fig. 2 (right), the root of the tree corresponds to level ALL with the single value *all*. The values of a certain dimension level L are found in the same level of the tree. Each node is characterized by a score for the preference concerning the combination of the non-context attributes with the context value of the node.

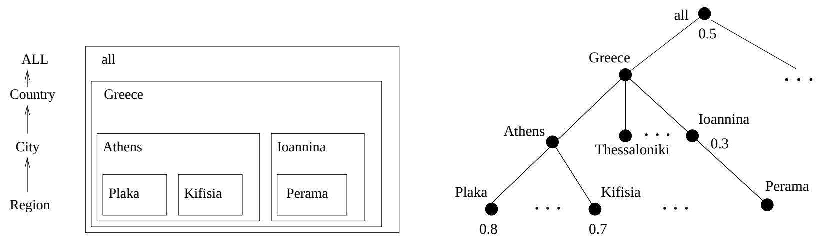


Fig. 2. Hierarchies on *location* (left) and the hierarchy tree of *location* (right).

If the context in a query refers to a level of the tree in which there is no explicit score given by the user, there are three ways to compute the appropriate score for a preference. In the first approach, we traverse the tree upwards until we find the first predecessor for which a score is specified. In this case, we assume that a user that defines a score for a specific level, implicitly defines the same score for all the levels below. In the second approach, we compute the average score of all the successors of the immediately following level, if such scores are available, else we follow the first approach. Finally, we can combine both approaches by computing a weighted average score of the scores from both the predecessor and the successors. In any case, we assume a default score of 0.5 at level *all*, if no score is given.

3 Storing Basic Preferences

We store basic user preferences in *hypercubes*, or simply *cubes*. The number of data cubes is equal to the number of context parameters, i.e., we have one cube for each context parameter. Formally, a *cube schema* is defined as a finite set of attributes $Cube = (C_i, A_1, \ldots, A_n, M)$, where C_i is a context parameter, $A_1, \ldots, A_n$ are non-context attributes and M is the interest score. The cubes for our running example are depicted in Fig. 3. In each cube, there is one dimension for the *points_of_interest*, one dimension for the users and one dimension for the context parameter. In each cell of the cube, we store the degree of interest for a specific preference.

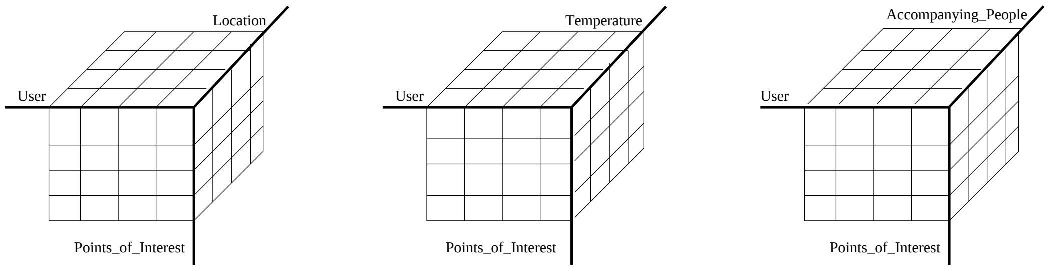


Fig. 3. Data cubes for each context parameter.

A relational table implements such a cube in a straightforward fashion. The primary key of the table is $C_i, A_1, \ldots, A_n$. If dimension tables representing hierarchies exist (see next), we employ foreign keys for the attributes corresponding to these dimensions. The schema for our running example which is based on the classical *star schema* is depicted in Fig. 4. There are three fact tables, $Temperature$, *Location* and *Accompanying_People*. The dimension tables are: $Users$ and *Points_of_Interest*. These are dimension tables for both fact tables.

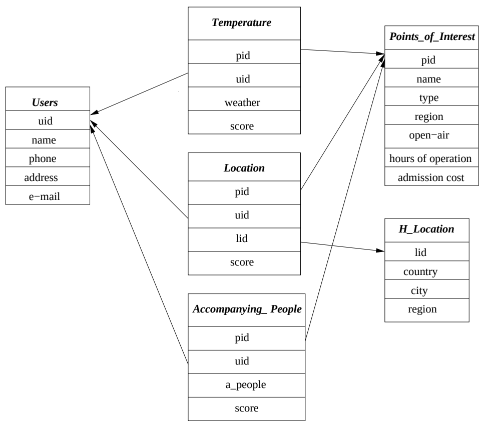
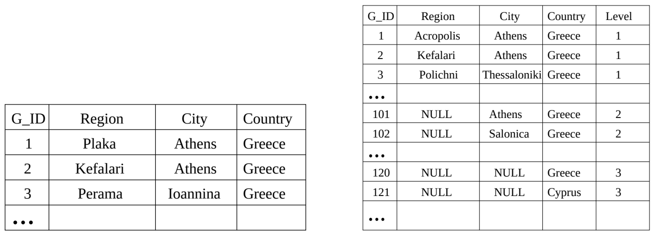


Fig. 4. The fact and dimension tables of our schema.

Regarding hierarchical context attributes, the typical way to store them is shown in Fig. 5 (left). In this modeling, we assign an attribute for each level in the hierarchy. We also assign an artificial key to efficiently implement references to the dimension table. The denormalized tables of this kind suffer from the fact that there exists exactly one row for each value of the lowest level of the hierarchy, but no rows explicitly representing values of higher levels of the hierarchy. Therefore, if we want to express preferences at a higher level of the hierarchy, we need to extend this modeling (assume for example that we wish to express the preferences of $Mary$ when she is in the city of $Thessaloniki$, independently of the specific region of $Thessaloniki$ she is found at). To this end, we use an extension of this approach, as shown in the right of Fig. 5. In this kind of dimension tables, we introduce a dedicated tuple for each value at any level of the hierarchy. We populate attributes of lower levels with $NULL$s. To explain the particular level that a value participates at, we also introduce a level indicator attribute. Dimension levels are assigned attribute numbers through a topological sort of the lattice.

To compute aggregate preferences from simple ones we need also to store the weights used in this computation. Weights are stored in a special purpose table $AggScores(w_{C1}, \ldots, w_{Ck}, A_{k+1})$. The value for each context parameter w_{Ci} is the weight for the respective interest score and the value A_{k+1} specifies the user who gives these weights. For instance, in our running example, the table $AggScores$ has the attributes $Location\_weight$, $Temperature\_weight$, $Accompanying\_People\_weight$, and $User$. A record in this table can be (*0.6, 0.3, 0.1, Mary*).



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

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

Fig. 5. A typical (left) and an extended dimension table (right).

Aggregate preferences are not explicitly stored. The main reason is space and time efficiency, since this would require maintaining a context cube for each context state and for each combination of non-context attributes. Assume that the context environment CE_X has n context parameters $\{C_1, C_2, \ldots, C_n\}$ and that the cardinality of the domain $dom(C_i)$ of each parameter C_i is (for simplicity) m. This means that there are m^n potential context states, leading to a very large number of context cubes and prohibitively high costs for their maintenance. Instead, we store only previously computed aggregate scores, using an auxiliary data structure (described in Section 4).

An advantage of using cubes to store user preferences is that they provide the capability of using *hierarchies* to introduce different levels of abstractions of the captured context data through the *drill-down* and *roll-up* operators [5]. The *roll-up* operation provides an aggregation on one dimension. Assume, for example, that the user has executed a query about Mary's most preferable point-of-interests in *Plaka*. However, this query has returned an unsatisfactory small number of answers. Then, Mary may decide that is worth broadening the scope of the search and investigate the broader *Athens* area for interesting places to visit. In this case, a *roll-up* operation on *location* can generate a cube that uses *cities* instead of *regions*. Similarly, *drill-down* is the reverse of roll-up and allows the de-aggregation of information moving from higher to lower levels of granularity.

4 Caching Context-Aware Queries

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

4.1 The Context Tree

Assume that the context environment CE_X has n context parameters $\{C_1, C_2, \ldots, C_n\}$. A way to store aggregate preferences uses the *context tree*, as shown in Fig. 6. There is one context tree per user. The maximum height of the context tree is equal to the number of context parameters plus one. Each context parameter is mapped to one of the levels of the tree and there is one additional level for the leaves. For simplicity, assume that context parameter C_i is mapped to level i. A path in the context tree denotes a *context state*, i.e., an assignment of values to context parameters. At the leaf nodes, we store a list of ids, e.g., *points_of_interest* ids, along with their aggregate scores for the associated context state, that is, for the path from the root leading to them. Instead of storing aggregate score values for all the ids, to be storage-efficient, we just store the $top - k$ ids (keys), that is the ids of the items having the k-highest aggregate scores for the path leading to them. The motivation is that this allows us to provide users with a fast answer with the data items that best match their query. Only if more than k-results are needed, additional computation will be initiated. The list of ids is sorted in decreasing order according to their scores.

The context tree is used to store aggregate preferences that were computed as results of previous queries, so that these results can be re-used by subsequent queries. Thus, it is constructed incrementally each time a context-aware query is computed. Each non-leaf node at level k contains cells of the form $[key, pointer]$, where *key* is equal to $c_{kj} \in dom(C_k)$ for a value of the context parameter C_k that appeared in some previously computed context query. The pointer of each cell points to the node at the next lower level (level $k+1$) containing all the distinct values of the next context parameter (parameter C_{k+1}) that appeared in the same context query with c_{kj}. In addition, *key* may take the special value *any*, which corresponds to the lack of the specification of the associated context parameter in the query (i.e., to the use of the special symbol '*').

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

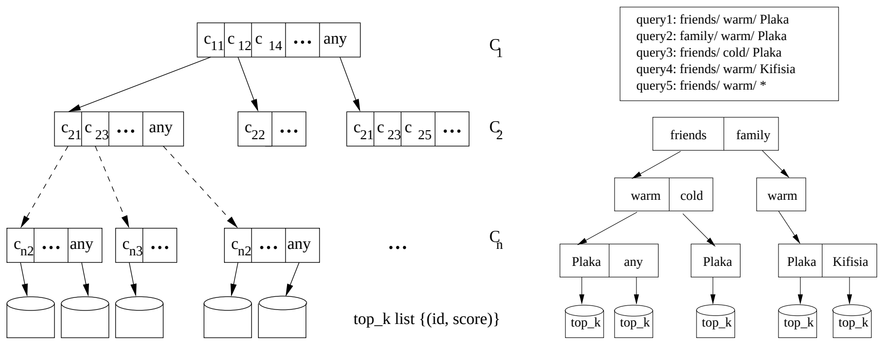

query1: friends/ warm/ Plaka
query2: family/ warm/ Plaka
query3: friends/ cold/ Plaka
query4: friends/ warm/ Kifisia
query5: friends/ warm/ *
C1
C2
Cn
any
top_k list {(id, score)}
friends
family
warm
cold
Plaka
Kifisia
top_k

Fig. 6. A context tree (left) and a set of aggregate preferences and the corresponding context tree (right).

– It is a directed acyclic graph with a single root node.
– There are at most $n+1$ levels, each one of the first n of them corresponding to a context parameter and the last one to the level of the leaf nodes.
– Each non-leaf node at level k maintains cells of the form $[key, pointer]$ where $key \in dom(C_k)$ for some value of c_k that appeared in a query or $key = any$. No two cells within the same node contain the same key value. The pointer points to a node at level $k+1$ having cells with key values which appeared in the same query with the key.
– Each leaf node stores a set of pointers to data sorted by their score.

For example, Fig. 6 (right) shows a set of context states expressed in five previously submitted queries and the corresponding context tree. Assume that the three context parameters are assigned to levels as follows: *accompanying_people* is assigned to the first level, *temperature* to the second and *location* to the third one. Leaf nodes store the ids of the $top-k$ *points_of_interest*, that is the places with the $top-k$ highest aggregate scores.

The context tree provides an efficient way to retrieve the *top-k* results that are relevant to a preference query. When a query is posed, we first check if there exists a context state that matches it in the context tree. If so, we retrieve the *top-k* results from the associated leaf node. Otherwise, we compute the answer and insert the new context state, i.e., the new path and the associated *top-k* results, in the tree. Thus a query is a simple traversal on the context tree from the root to a leaf. At level i, we search a node for a cell having as key value the i^{th} value of the query and descend to the next level following the appropriate pointer. For a context tree with n context parameters $(C_1, C_2, \ldots, C_n)$, if each parameter has $|dom(C_i)|$ values in its domain, the maximum number of cells that are required to be visited for a query is $|dom(C_1)| + |dom(C_2)| + \ldots + |dom(C_n)|$, while the number of nodes is equal to the height of the tree.

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.

Finally, there are two additional issues related to managing the context 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 is the least frequently or the least recently used one. 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 context 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 context 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.

4.2 Querying with Approximate Results

We consider ways of extending the use of the context tree to not only providing answers in the case of queries in exactly the same context state, but also providing approximate answers to queries whose context state is "similar" to a stored one. One such case involves the '*' operator. If the value of some context parameter is "any" (i.e., '*'), we check whether results for enough values of this parameter are already stored in the tree. In particular, if the number of the existing values of this parameter in the same node of the context tree is larger than a threshold value, we do not compute the query from scratch, but instead, merge the stored results for the existing values of the parameter. We call this threshold, *coverage approximation threshold* (*ct*). Its value may be either system defined or given as input by the user.

Another case in which we can avoid recomputing the results of a query is when the values of its context parameters are "similar" with those of some stored context state. For example, if one considers the near-by locations *Thisio* and *Plaka* as similar, then the query *friends/warm/Thisio* can use the results associated with the stored query *friends/warm/Plaka* (Fig. 6 (right)).

To express when two values of a context parameter are similar, we introduce a *neighborhood approximation threshold* (*nt*). In particular, two values c_i and c'_i of a context parameter C_i are consider similar up to nt_i, if and only if for any tuple t with score d_i for $C_i = c_1$ and d'_i for $C_i = c'_i$, it holds:

$$|d_i - d'_i| \leq nt_i \tag{1}$$

for a small constant nt, $0 \leq nt \leq 1$.

The threshold nt_i may take different values for each context parameter C_i depending for instance, on the type of its domain. As before, the threshold may be either determined by the user or the system. To estimate the quality of an approximation, we are interested in how much the results for two queries in two similar context states differ, that is how much different is the rating of the results in the two states, thus leading to a different set of *top-k* answers. We have proved [4] the following intuitive property that states that for any two tuples, the difference between their aggregate scores in two states, s and s' that differ only at the value of one context parameter, C_i, is bounded, if the two values of C_i are similar. This indicates that the relative order of the results in states s and s' is rather similar.

Property 1. Let t_1, t_2 be two tuples that have aggregate scores d_1, d_2 in a context state s and d'_1, d'_2 in a context state s' respectively. If s, s' differ only in the values of one context parameter, C_i, and these two values of C_i are similar up to nt_i, then if $|d_1 - d_2| \leq \varepsilon$, $|d'_1 - d'_2| \leq \varepsilon + 2 * w_1 * nt$, where w_i is the weight of the context parameter C_i.

Property 1 is easily generalized for the case in which two states differ in more than one similar up to nt_i context parameter. In particular:

Property 2. Let t_1, t_2 be two tuples that have aggregate scores d_1, d_2 in a context state s and d'_1, d'_2 in a context state s' respectively. If s, s' differ only in the value of m context parameters, C_{j_k}, $1 \leq k \leq m$, and these two values of C_{j_k} are similar up to nt_{j_k}, then if $|d_1 - d_2| \leq \varepsilon$, then, $|d'_1 - d'_2| \leq \varepsilon + 2 * (w_{j_1} * nt_{j_1} + w_{j_2} * nt_{j_2} \ldots + w_{j_m} * nt_{j_m})$, where w_{j_k} is the weight of a context parameter C_{j_k}.

5 Performance Evaluation

In this section, we evaluate the expected size of the context tree as well as the accuracy of the two approximation methods. We divide the input parameters into three categories: *context parameters*, *query workload parameters*, and *query approximation parameters*. In particular, we use three context parameters and thus, the context tree has three levels (plus one for the $top-k$ lists). There are two different types regarding the *cardinalities* of the domains of the context parameters: the *small* domain with 10 values and the *large* one with 50 values.

We performed our experiments with various numbers of queries stored at the context tree varying from 50 to 200, while the number of tuples is 10000. 10% of the values in the queries are '*'. The other 90% are either selected *uniformly* from the domain of the corresponding context parameter, or follow a *zipf* data distribution. The *coverage approximation threshold ct* refers to the percentage of values that need to be stored for a context parameter, to compute the $top-k$ list by combining their corresponding $top-k$ lists when there is the '*' value at the corresponding level in a new query. The *neighborhood approximation threshold nt* refers to how similar are the scores for two "similar" values of a context parameter. Our input parameters are summarized in Table 1.

5.1 Size of the Context Tree

In the first set of experiments, we study how the mapping of the context parameters to the levels of the context tree affects its size. In particular, we count the total number of cells in the tree as a function of the number of stored queries, taking into consideration the different orderings of the parameters. For a context tree with three parameters, we call *ordering 1* the ordering of the context parameters in which the parameter whose domain has 10 values is assigned to the first level, the parameter with 10 to the second one, and the parameter with 50 values to the last one. *Ordering 2* is the ordering when the domains have 10, 50, 10 values respectively, and for the *ordering 3* the domains have 50, 10, 10 values. As discussed in Section 3, the mapping of the context parameters to levels that is expected to result in a smaller sized tree, is the one that places the context parameters with the large domains lower in the tree.

Table 1. Input Parameters

Context Parameters	**Default Value**	**Range**
Number of Context Parameters	3	
Cardinality of the Context Parameters Domains		
Small	10	
Large	50	
Query Workload		
Number of Tuples	10000	
Number of Stored Queries		50-200
Percentage of '*' values	10%	
Data Distributions	$uniform$	
	zipf - a = 1.5	a = 0.0 - 3.5
Top-k results	10	
Query Approximation		
Coverage Approximation Threshold (ct)	$\geq$ 40%, 60%, 80%	
Neighborhood Approximation Threshold (nt_i)	0.08	0.04, 0.08, 0.12
Weights	0.5, 0.3, 0.2	

In our experiments, 10% of the query values are selected to be the *any* value. The rest 90% of the values are selected from the corresponding domain, either using a *uniform* data distribution, or a *zipf* data distribution with $a = 1.5$. In both cases, as shown in Fig. 7, the total storage space is minimized when the parameter with the large domain (50 values) is assigned to the last level of the tree (ordering 3). Also, for the $zipf$ distribution (Fig. 7 (right)), the total number of cells is smaller than for the $uniform$ distribution, (Fig. 7 (left)), because using the $zipf$ distribution "hot" values appear more frequently in queries, i.e., more context values are the same.

However, the best way of assigning parameters to levels depends also on the query workload, that is, on the percentage of values from the domain of each parameter that *actually* appears in the queries. 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. This is shown with the next experiment (Fig.

8). We performed this experiment 50 times with 200 queries. The values of the context parameters with *small* domains are selected using a $uniform$ data distribution and the values of the context parameter with the *large* domain are selected 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).

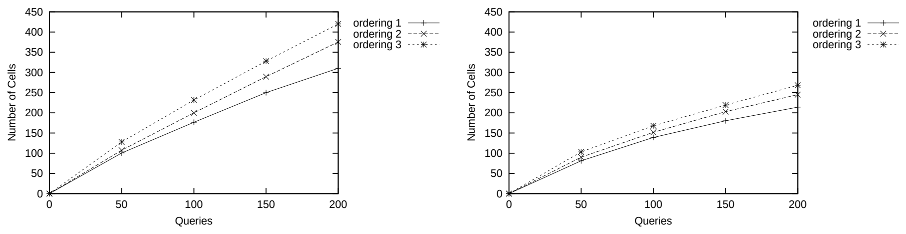


Fig. 7. Uniform (left) and zipf data distribution with $a = 1.5$ (right).

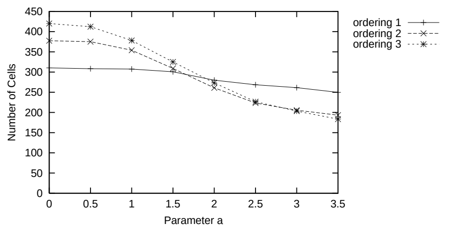


Fig. 8. Combined $uniform$ and $zipf$ data distributions.

5.2 Accuracy of the Approximations

In this set of experiments, we evaluate the accuracy of the approximation when using the coverage and the neighborhood approximation thresholds. In both cases, we report how many of the *top-k* tuples computed using the results stored in the tree actually belong to the *top-k* results.

Using the Coverage Approximation Threshold. A *coverage approximation threshold* of *ct*% means that at least *ct*% of the required values are available, i.e., are already computed and stored in the context tree. We use three values for *ct*, namely, 40%, 60%, and 80%. All weights take the value 0.33. In Fig. 9, we present the percentage of different results in the *top-k* list for each *ct* value, when a '*' value is given for a parameter with a small domain or a large domain, respectively. To compute the actual aggregate preference scores, we use the second of the two approaches presented in Section 2.2. The approximation is better when '*' refers to the context parameter with the large domain. This happens because in this case, more related paths of the context tree are available, and so, more *top-k* lists of results are merged to produce the new *top-k* list.

Using the Neighborhood Approximation Threshold. We consider first that a query is similar with another one, when they have the same values for all the context parameters except one, and the values of this parameter are similar up to *nt*. We use three values for the parameter *nt*: 0.04, 0.08, and 0.12. The weights have the values 0.5, 0.3, and 0.2. We count first, the number of different results between two similar queries that differ at the value of one context parameter, as a function of the weight of this parameter, taking into consideration the different values of *nt*. The results are depicted in Fig. 10 (left). Then, we examine the case in which the values of two context parameters are different (Fig. 10 (right)). In this case, the accuracy of the results depens on both the weights that correspond to the context parameters whose values are similar. As expected, the smaller the value of the parameter *nt*, the smaller the difference between the results in the *top-k* list of two similar queries. Note further, that the value of the weight that corresponds to the similar context parameter also affects the number of different results: the smaller the value of the weight, the smaller the number of different results.

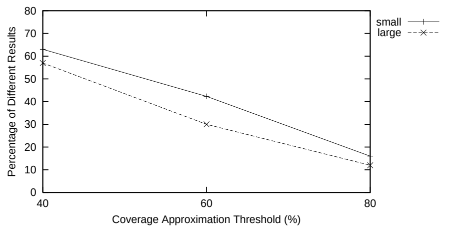


Fig. 9. Different results when the *ct* has values 40%, 60%, 80%.

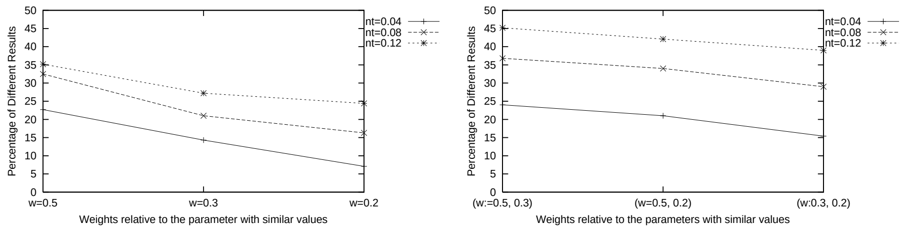


Fig. 10. Different results between two similar queries when $nt = 0.04, 0.08, 0.12$.

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 [6] and the Dartmouth Solar System [7]), there has been only little work on the integration of context into databases. Next, we discuss work related to context-aware queries and preference queries. A preliminary version of the model without the context tree and the performance evaluation part appears in [8].

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 [9], 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 [5] for developing context-aware applications that 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. In our work, we use data cubes for storing context-dependent preferences and answering queries. The Context Relational Model (CR) [10] 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 [11] where the problem of representing context-dependent semistructured data is studied. A similar context model is also deployed in [12] for enhancing web service discovery with contextual parameters.

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 [13], since it is more easy for users to employ. In the quantitative framework of [1], 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, [14]), 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

The use of context allows users to receive only relevant information. In this paper, we consider integrating context in expressing preferences, so that when a user poses a preference query in a database, the result also depends on context. In particular, each user indicates preferences on specific attribute values of a relation. Such preferences depend on context and are stored in data cubes. To allow re-using results of previously computed preference queries, we introduce a hierarchical data structure, called context tree. This tree can be used further to produce approximate results, using similar stored results. Our future work includes exploring context information in answering additional queries, not just preference ones.

References

1. Koutrika, G., Ioannidis, Y.: Personalization of Queries in Database Systems. In: Proc. of ICDE. (2004)
2. Dey, A.K.: Understanding and Using Context. Personal and Ubiquitous Computing **5** (2001)
3. Chen, G., Kotz, D.: A Survey of Context-Aware Mobile Computing Research. Dartmouth Computer Science Technical Report TR2000-381 (2000)
4. Stefanidis, K., Pitoura, E., Vassiliadis, P.: Modeling and Storing Context-Aware Preferences (extended version). University of Ioannina, Computer Science Departement, TR 2006-06 (2006)
5. Harvel, L., Liu, L., Abowd, G.D., Lim, Y.X., Scheibe, C., Chathamr, C.: Flexible and Effective Manipulation of Sensed Contex. In: Proc. of the 2nd Intl. Conf. on Pervasive Computing. (2004)
6. Salber, D., Dey, A.K., Abowd, G.D.: The Context Toolkit: Aiding the Development of Context-Enabled Applications. CHI Conference on Human Factors in Computing Systems (1999)
7. Chen, G., Li, M., Kotz, D.: Design and implementation of a large-scale context fusion network. International Conference on Mobile and Ubiquitous Systems: Networking and Services (2004)
8. Stefanidis, K., Pitoura, E., Vassiliadis, P.: On Supporting Context-Aware Preferences in Relational Database Systems. International Workshop on Managing Context Information in Mobile and Pervasive Environments (2005) (extended version to appear in JPCC).
9. Feng, L., Apers, P., Jonker, W.: Towards Context-Aware Data Management for Ambient Intelligence. In: Proc. of the 15th Intl. Conf. on Database and Expert Systems Applications (DEXA). (2004)
10. Roussos, Y., Stavrakas, Y., Pavlaki, V.: Towards a Context-Aware Relational Model. In the proceedings of the International Workshop on Context Representation and Reasoning (CRR'05) (2005)
11. Stavrakas, Y., Gergatsoulis, M.: Multidimensional Semistructured Data: Representing Context-Dependent Information on the Web. International Conference on Advanced Information Systems Engineering (CAiSE 2002) (2002)
12. Doulkeridis, C., Vazirgiannis, M.: Querying and Updating a Context-Aware Service Directory in Mobile Environments. Web Intelligence (2004) 562–565
13. Agrawal, R., Wimmers, E.L.: A Framework for Expressing and Combining Preferences. In: Proc. of SIGMOD. (2000)
14. Chomicki, J.: Preference Formulas in Relational Queries. TODS **28** (2003)