A Context-Aware Preference Database System

Kostas Stefanidis
Department of Computer Science, University of Ioannina, Greece, kstef@cs.uoi.gr
Evaggelia Pitoura
Department of Computer Science, University of Ioannina, Greece, pitoura@cs.uoi.gr
Panos Vassiliadis
Department of Computer Science, University of Ioannina, Greece, pvassil@cs.uoi.gr

Received: January XX 2005; revised: November XX 2005

***Abstract*— A context-aware system is a system that uses context to provide relevant information or services to its users. While there has been a variety of context middleware infrastructures and context-aware applications, little work has been done on integrating context into database management systems. In this paper, we consider a preference database system that supports context-aware queries, that is, queries whose results depend on the context at the time of their submission. We propose using data cubes to store the dependencies between context-dependent preferences and database relations and OLAP techniques for processing context-aware queries. This allows for the manipulation of the captured context data at various levels of abstraction, for instance, in the case of a context parameter representing location, preferences can be expressed for example at the level of a city, the level of a country or both. To improve query performance, we use an auxiliary data structure, called context tree. The context tree stores results of past context-aware queries indexed by the context of their execution. Finally, we outline the implementation of a prototype context-aware restaurant recommender.**

***Index Terms*— context, preference, OLAP, context-awareness, querying processing**

I. Introduction

Context is any information that can be used to characterize the situation of an entity. An entity is a person, place or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves [1]. There are various types of context including time, location, and available computing and communication resources. A system is *context-aware*, if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task. Although there has been a lot of work on developing a variety of context infrastructures and context-aware middleware and applications (for example, the Context Toolkit [2] and the Dartmouth Solar System [3]), there has been only little work on integrating context information into databases. Most of this work has focused on a particular type of context, that of location, mainly in the context of moving object databases.

In this paper, we investigate the use of context in relational database management systems. We consider *context-aware* queries which are queries whose results depend on the context at the time of their submission. In particular, users express their preferences on specific attributes of a relation. Such preferences depend on context, that is, they may have different values depending on context.

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

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

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

As an example, consider a database schema with information about restaurants and users (Fig. 1). In this application, we consider two context parameters as relevant: $location$ and $weather$. Users have preferences about restaurants that they express by providing a numerical score between 0 and 1 that quantifies their degree of interest for a restaurant. The degree of interest of a user for a restaurant depends on the values of the two relevant context parameters. For instance, a user may want to eat different kinds of food depending on the current weather conditions. For example, user $Mary$ may give to restaurant *Zoloushka* that serves "Russian" food a higher score when the weather is $rainy$ than when the weather is $sunny$. Furthermore, the current user's location affects the result of a query, for example, a user may prefer restaurants that are nearby her current location. The user provides such preference scores that depend on a single context parameter, in this example, preference scores that depend on location and preference scores that depend on weather. These basic preferences are then combined to produce an aggregate score that depends on more than one context parameter. In addition, a user can specify preferences without giving values for all context parameters. In particular, the special value '*' for a context parameter denotes that the user preference does not depend on it.

We store simple preferences in data cubes, following the

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

Aggregate preferences are not explicitly stored. To improve performance, we propose storing aggregate preferences computed as results of previous queries using an auxiliary data structure called *context tree*. A path in the context tree corresponds to an assignment of values to context parameters, that is, to a context state, for which the aggregate score has been previously computed. Results stored in a context tree are re-used to speed-up query processing. We also show how search in the context tree can be improved using a variation of a Bloom-based filter for testing membership in the tree.

As a proof-of-concept, we have implemented a simple application that allows users to express their preferences regarding our running example of a restaurant database. These preferences depend on two context parameters, location and weather. Users can pose preference queries whose results depend on context.

Contributions. Summarizing, we make the following contributions:

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

Paper Organization. The rest of this paper is structured as follows. Section II introduces our preference model, while Section III focuses on how preferences are stored. Section IV discusses query processing in our framework. Section V introduces the context tree for storing aggregate preferences. Our prototype implementation is outlined in Section VI, while related work is presented in Section VII. Section VIII concludes the paper with a summary of our contributions.

II. A Logical Model for Context and User Preferences

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

A. Modeling Context

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

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

Attributes and Relations. As usual, we assume a countable collection of attribute names. Each attribute A_i is characterized by a name and a domain $dom(A_i)$. A relation schema is a finite set of attributes and a relation instance is a finite subset of the Cartesian product of the domains of the relation schema [4].

Context Parameters. 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 C_X as a set of n context parameters $\{c_1, c_2, \ldots, c_n\}$.

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

Hierarchies for Attributes. It is possible for an attribute to participate in an associated *hierarchy of levels* of aggregated data i.e., it can be viewed from different levels of detail. Formally, an *attribute hierarchy* is a lattice of attributes – called *levels* for the purpose of the hierarchy – $L = (L_1, \ldots, L_m, ALL)$. We require that the upper bound of the lattice is always the level ALL, so that we can group all the values into the single value all. The lower bound of the lattice is called the detailed level of the parameter. For instance, let us consider the hierarchy $location$ of Fig. 2. Levels of $location$ are $Region$, $City$, $Country$, and ALL. $Region$ is the most detailed level. Level ALL is the most coarse level for all the levels of a hierarchy. Aggregating to the level ALL of a hierarchy ignores the respective parameter in the grouping (i.e., practically groups the data with respect to all the other parameters, except for this particular one).

The relationship between the values of the context levels is achieved through the use of the set of $anc_{L_1}^{L_2}$ functions. A function $anc_{L_1}^{L_2}$ assigns a value of the domain of L_2 to a value of the domain of L_1. For instance, $anc_{Region}^{City}(Acropolis) = Athens$. A formal definition of these hierarchies can be found in [5].

Dynamic and Static Context Parameters. We distinguish between two kinds of context parameters: (a) static and (b) dynamic context parameters. *Static context parameters* take

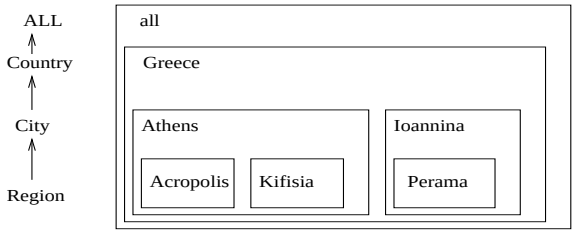

Fig. 2. Hierarchies on $location$.

as value a simple value out of their domain. *Dynamic context parameters* on the other hand, are instantiated by the application of a function, the result of which is an instance of the domain of the context parameter. In our example, we assume that $weather$ is a static parameter, i.e., each new value for $weather$ is derived by an explicit update. On the other hand, $location$ is a dynamic parameter. In particular, $location$ is defined as a function of time and in that way, we can compute the value of this parameter at the point we want to use it, without the need for continuous explicit updates. Defining appropriate functions and procedures for determining the value of a dynamic context parameter in the current or some future time instant is beyond the scope of this paper. There has been related work in the context of managing the location of moving objects [6].

B. Contextual Preferences

In this section, we define how a context state affects the results of a query. In our model, each user expresses his/her preference by providing a numerical score between 0 and 1 [7]. This score expresses a degree of interest, which is a real number. Value 1 indicates extreme interest. In reverse, value 0 indicates no interest for a preference. The special value '$\oslash$' for a preference means that there is a user's veto for the preference. Furthermore, the value '$*$' represents that any value is acceptable.

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

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

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

In our reference example (Fig 1), there are two context parameters, $location$ and $weather$. Also, the set of non-context parameters are attributes about $restaurants$ and $users$. Assume a user $Mary$ and a restaurant called $BeauBrummel$ located near $Acropolis$ in $Athens$ that serves $French$ cuisine. $Mary$ likes to eat $French$ cuisine when the weather is $cloudy$, so she assigns high scores to $BeauBrummel$ when she is in $Acropolis$ and the weather is $cloudy$ expressed by the following basic preferences:

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

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

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

The interest score of the aggregate preference is a *value function* of the individuals scores (the degrees of the basic preferences). The value function prescribes how to combine basic preferences to produce the aggregate score, according to the user's profile. In this paper, we assume that value functions are based on a weighted average of the simple preferences. Users define in their profile how the basic scores contribute to the aggregate ones, by giving a weight to each context parameter. So, if the weight for a context parameter is w_i and $interest\_score_i$ is the score defined by the associated basic preference, then the aggregate interest score will be:

$$interest\_score =$$
$$w_1 \times interest\_score_1 + \ldots + w_k \times interest\_score_k.$$

For instance, in the previous example if the weight of $location$ is 0.6 and the weight of $weather$ is 0.4, the preference has score: $0.6 \times 0.8 + 0.4 \times 0.9 = 0.84$ (from the above value function). That is, we have:

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

Table I summarizes all notations used in our model.

TABLE I
NOTATIONS

Name	Notation
Attribute	A_i
Domain of A_i	$dom(A_i)$
Context parameter	c_i
Context environment for an application X	C_X
Context state at time instant t	$CS(t)$
Weight for a context parameter c_i	w_i

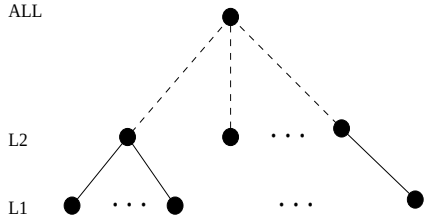

Fig. 3. The hierarchy tree for parameter L.

C. Inheriting Preferences

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

The tree of Fig. 3 represents the different levels of hierarchy for a context parameter. For the parameter L, let L_1, L_2,..., L_m, ALL be the different levels of the hierarchy, which can take various different values. There is a hierarchy tree, for

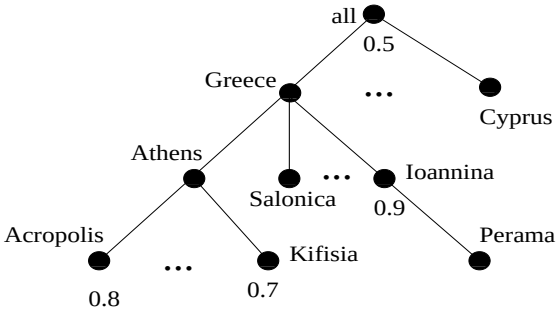

Fig. 4. The hierarchy tree of location.

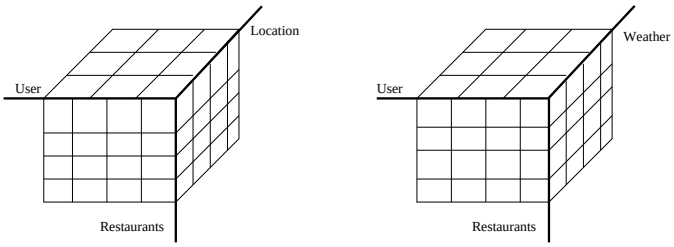

Fig. 5. Data cubes for each context parameter.

each combination of non-context parameters. In our reference example (Fig. 4), there is a hierarchy tree for each user profile and for a specific restaurant that represents the interest scores of the user for the restaurants, accordingly to the context parameter's hierarchy. The root of the tree concerns level ALL with the single value all. The values of a certain dimension level L are found in the same level of the tree (e.g., $Athens$ and $Ioannina$, being both members of the dimension level $City$, are found at the same level of the tree in Fig. 4). The ancestor relationships $anc_{L_1}^{L_2}$ are translated to parent-child relationships in the tree (e.g., the node $Greece$ is the parent of the node $Athens$). Each node is characterized by a score value for the preference concerning the combination of the non-context attributes with the context value of the node.

If the query conditions refer to a level of the tree in which there is no explicit score given by the user, we propose three ways to find the appropriate score for a preference. In the first approach, we traverse the tree upwards until we find the first predecessor for which a score is specified. In this case, we assume that, a user that defines a score for a specific level, implicitly defines the same score for all the lower levels. In the second approach, we compute the average score of all the successors of the immediately lower level. Finally, following a hybrid approach, we can compute a weighted average score combining the scores from both the predecessor and the successors. In any of the above cases, if no score is defined at any level of the hierarchy, there is a default score of 0.5 for value all.

Take for example, Fig. 4 that depicts a hierarchy for a $user$ ($Mary$) and a $restaurant$ ($BeauBrummel$). So, for instance the restaurant *Beau Brummel* has score 0.8 when $Mary$ is near $Acropolis$, 0.7 when she is in $Kifisia$, and 0.9 when she is in $Ioannina$. The root of the hierarchy has the default score 0.5. These degrees of interest scores, except the last one, have been explicitly defined by the user in her profile. If the query conditions refer to *Athens*, for which there is no score, the first approach gives score 0.5, because this is the first available predecessor's score. If we choose the second approach, this leads to score $(0.8 + 0.7)/2 = 0.75$, while the third one produces a weighted combination of the above scores.

D. Discussion

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.

Finally, since the focus of this work is on efficiently combining preferences and database operations, our working assumption is that preferences are explicitly specified by users. Alternatively, preferences may be deduced by the previous behavior of the user, for instance by using data mining techniques on the history of the user database accesses. The issue of implicitly inferring preferences is orthogonal to the work presented in this paper. There has been some previous work on the topic [8], that can be integrated in our approach.

III. The Storage Model

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

A. Storing Basic Preferences

There is a straightforward way to store our context and preference information in the database. We organize preferences as data cubes, following the OLAP paradigm [5]. In particular, we store basic user preferences in $hypercubes$, or simply, $cubes$. The number of data cubes is equal with the number of context parameters, i.e., we have one cube for each parameter, as shown in Fig. 5. In each cube, there is a dimension for restaurants, a dimension for users and a dimension for the context parameter. In each cell of the cube, we store the degree of interest for a specific preference. This way, we maintain the score for a user, a restaurant and a context parameter. Formally, a $cube$ is defined as a finite set of attributes $C = (c_i, A_1, \ldots, A_n, M)$, where c_i is a context parameter, $A_1, \ldots, A_n$ are non-context attributes and M is the interest score. The values of a cube are the values of the corresponding preference rules. A relational table implements such a cube in a straightforward fashion. The primary key of the table is $c_i, A_1, \ldots, A_n$. If there exist dimension tables representing hierarchies (see next), we employ foreign keys for the attributes corresponding to these dimensions.

Our schema which is based on the classical star schema is depicted in Fig. 6. As we can see, there are two fact tables,

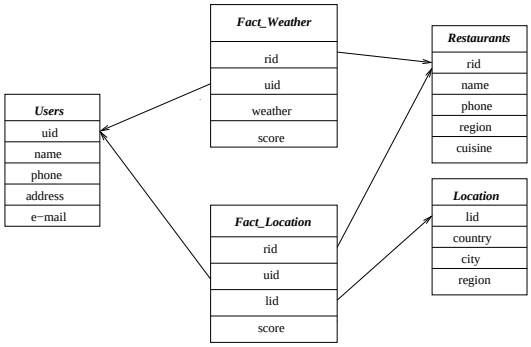

Fig. 6. The two fact tables of our schema (one for each context parameter) and the dimension tables for $Users$ and $Restaurants$.

$Fact\_Location$ and $Fact\_Weather$. The dimension tables are: $Users$ and $Restaurants$. These are dimension tables for both fact tables.

B. Storing Context Hierarchies

An advantage of using cubes to store user preferences is that they provide the capability of using $hierarchies$ to introduce different levels of abstractions of the captured context data [9]. In that way, we can have a hierarchy on a given context dimension. Context dimension hierarchies give the opportunity to the application to use a combination of data between the fact and the dimension tables on one of the context parameters. The typical way to store data in databases is shown in Fig. 7 (left). In this modeling, we assign an attribute for each level in the hierarchy. We also assign an artificial key to efficiently implement references to the dimension table. The contents of the table are the values of the $anc_{L_1}^{L_2}$ functions of the hierarchy. The denormalized tables of this kind, participating in a database schema (often called a *star schema*) suffer from the fact that there exists exactly one row for each value of the lowest level of the hierarchy, but no rows explicitly representing values of higher levels of the hierarchy. Therefore, if we want to express preferences at a higher level of the hierarchy, we need to extend this modeling (assume for example that we wish to express the preferences of $Mary$ when she is in $Cyprus$, independently of the specific region, or city of Cyprus she is found at).

To this end, in our model, we use an extension of this approach, as shown in the right of Fig. 7. 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.

C. Storing the Value Functions

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

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

The non-context parameters pin the values of the aggregate scores to specific numbers and then, the individual scores for each context parameter are collected from each context table. Recall that the formula for computing an aggregate preference is: $interest\_score = w_1 \times interest\_score_1 + \ldots + w_k \times interest\_score_k$.

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

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

Fig. 7. A typical (left) and an extended dimension table (right).

Therefore, the only extra information that needs to be stored concerns the weights employed for the computation of the formula. To this end, we employ a special purpose table $AggScores(w_{c_1}, \ldots, w_{c_k}, A_{k+1}, \ldots, A_n)$. The value for each context parameter w_{c_i} is the weight for the respective interest score and the value for each non-context attribute A_j is the specific value uniquely determining the aggregate preference. For instance, in our running example, the table $AggScores$ has the attributes $Location\_weight$, $Weather\_weight$ and $User$ and $Restaurant$. A record in this table can be *(0.6, 0.4, Mary, Beau Brummel)*. Assume that from $Mary's$ profile, we know that *Beau Brummel* has interest score at the current location 0.8 and at the current weather 0.9, then, the aggregate score is: $0.6 \times 0.8 + 0.4 \times 0.9 = 0.84$.

D. Storing Aggregate Preferences

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

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

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

IV. QUERYING CONTEXT

In this section, we classify the query operations that can be posed to our context-aware DBMS, by exploiting the combined information on preferences and context.

A. Queries with Basic Preferences

Firstly, there are queries executed without the need for the computation of an aggregate score. In this category of queries, users explicitly define that they are not interested in specific context parameters. For example, the following query computes the users' preferences directly.

Query 1 *Look for $Mary's$ most preferable restaurants near Acropolis, independently of the status of weather.*
In SQL, the query is:

- *SELECT* $R.name$, $FL.score$
  FROM Users U, Restaurants R, $Fact\_Location$ *FL,* $Location$ *L*
  WHERE $U.uid = FL.uid$ *AND* $R.rid = FL.rid$ *AND* $L.lid = FL.lid$ *AND* $U.name = 'Mary'$ *AND* $L.location = 'Acropolis'$
  ORDER BY FL.score DESC;

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

B. Queries with Aggregate Preferences

Another type of queries involves the computation of aggregate scores from simple ones. For example, the following query needs to compute an aggregate score:

Query 2 *Look for $Mary's$ most preferable restaurants (in the current context).*

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

- *SELECT* $R.name$, $FL.score$
  FROM Users U, Restaurants R, $Fact\_Location$ *FL,* $Location$ *L*
  WHERE $U.name = 'Mary'$ *AND* $U.uid = FL.uid$ *AND* $R.rid = FL.rid$ *AND* $L.lid = FL.lid$ *AND* $current\_location = 'Acropolis'$*;*
  and
- *SELECT* $R.name$, $FW.score$
  FROM Users U, Restaurants R, $Fact\_Weather$ *FW*
  WHERE $U.name = 'Mary'$ *AND* $U.uid = FW.uid$ *AND* $R.rid = FW.rid$ *AND* $current\_weather = 'sunny'$*;*

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

C. Computing Aggregate Scores

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

1) First, we select specific values for $Users$ and for the context parameter. For instance, for the first cube a selection could be on a value of $location$, e.g., $Acropolis$ and for a value of $user$, e.g., $Mary$.
2) Second, having pinned all dimension attributes to a specific value, we have all the preference interest scores available. In fact, the individual scores for each context parameter are collected from each context table (although this practically involves a relational join on all non-context parameters, it is quite more easy to simply collect the values from the respective cubes from a set of point queries over them). So, we can compute the aggregate score of a preference by using a value function (as described in the previous sections).
3) In the context of an OLAP session, the aggregate scores just computed for a user can be stored in a new transient cube. As with cubes concerning basic preferences, a cube concerning aggregate preferences has one attribute for each context and non-context parameter and an extra attribute for the interest score. Then, the user can reuse the result of a query, by just using the last cube, without executing all the above steps. In Section V, we describe a space-efficient structure for storing such results.

D. Traditional OLAP operators

OLAP provides a principled way of querying information. The traditional techniques for relational querying are enriched with special purpose query operators, such as roll-up and drill-down [10], [5].

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

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

Query 3

- *SELECT* $R.name$, $FL.score$
  FROM Users U, Restaurants R, $Fact\_Location$ *FL,* $Location$ *L*
  WHERE $U.uid = FL.uid$ *AND* $R.rid = FL.rid$ *AND* $L.lid = FL.lid$ *AND* $U.name = 'Mary'$ *AND*

$L.city = 'Athens'$
ORDER BY FL.score DESC;

Drill-down. Similarly, *drill-down* is the reverse operation, i.e., when we have the result of a query which includes $restaurants$ that are located in $Athens$, we can take a result that includes $restaurants$ located at $Acropolis$, using the *drill-down* operator.

V. Caching Context States

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

A. The Context Tree

Assume that the context environment C_X has n context parameters $\{c_1, c_2, \ldots, c_n\}$. An alternative way to store aggregate preferences uses a *context tree*, as shown in Fig. 8. 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 ones. There is one context tree per user or per system-defined profile (i.e., per group of users with similar interests, see Section II-D). The maximum height of the context tree is equal to the number of context parameters plus one. Each context parameter is mapped onto one of the levels of the tree and there is one additional level for the leaves. For simplicity, assume that context parameter c_i is mapped to level i. A path from the root to a leaf of the context tree corresponds to a *context state*, i.e., an assignment of values to context parameters.

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

The context tree is constructed incrementally each time a context-aware query is computed. Each non-leaf node at level k contains cells of the form $[key, pointer]$, where key is equal to $c_{kj} \in dom(c_k)$ for a value of the context parameter c_k that appeared in some previously computed context query. The pointer of each cell points to the node at the next lower level (level $k+1$) containing all the distinct values of the next context parameter (parameter c_{k+1}) that appeared in the same context query with c_{kj}. In addition, key may take the special value any, which corresponds to the lack of the specification of the associated context parameter in the query.

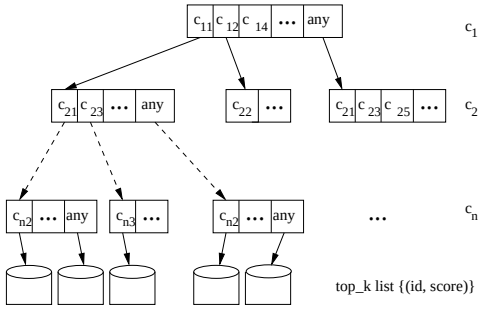

Fig. 8. A context tree.

For example, assume two context parameters, $location$ and $weather$ and that $weather$ is assigned to level m of the tree and $location$ to the level just below it, level m + 1. Then, take for instance a query, where the user specifies $weather = cloudy$, but gives no value for location. Then, there will be a cell $[cloudy, pointer]$ at level m pointing to a node at level $m+1$ containing a cell $[any, pointer]$. Initially, the context tree is empty, that is the root node contains a single cell of the form $[any, null]$.

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

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

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

In Fig. 9, we present a set of context preferences as expressed in four previously submitted queries. Assume again that we have two context parameters, $weather$ and $location$ and for the sake of the example that $weather$ is assigned to the first level of the tree and $location$ to the next one. Leaf nodes store the ids of the top-k restaurants, that is the restaurants with the top-k highest aggregate scores. For these preference queries, the context tree of Fig. 10 is constructed as follows.

For the first preference ($cloudy/Plaka$), the first path of the tree (the leftmost one) is constructed. The next preference ($cloudy/Acropolis$) has the same value for the context parameter $weather$ with the first one, so we do not add any new cell to the root node. Next, we add a cell for $Acropolis$ to the node that the root points to. The third preference ($sunny/Plaka$) has value $sunny$ for $weather$ and this leads to the creation of a new cell in the root node. As before, the last preference

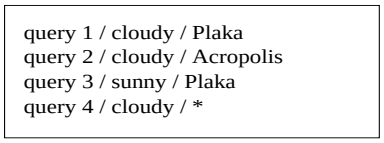

Fig. 9. A set of aggregate preferences.

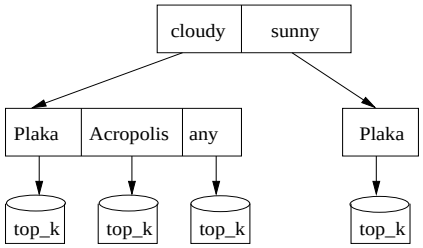

Fig. 10. The context tree for the preference of Fig 9.

$(/cloudy/*)$ has a $cloudy$ value and so, the remaining path for that preference has as a predecessor the $cloudy$ cell of the root. The any cell is added at the corresponding node of the second level for the $*$ operator of this preference.

When a query is issued, we first check whether there exists a context state that matches it in the context tree. If so, we retrieve the top-k results from the associated leaf node. Otherwise, we compute the answer and insert the new context state in the tree. There is a number of interesting variations. For instance, instead of storing the results of all queries, we may just store the results of the most frequently requested ones. This can be easily implemented by associating a counter with each path and replacing (deleting) from the tree the path that is less frequently used.

The context tree resembles the Dwarf data structure [11] used to compute and store data cubes. Whereas Dwarf is built by scanning the fact table and includes all existing combinations of values for all cube dimensions, the context tree is incrementally computed, each time a preference query is evaluated and includes only paths (context states) previously queried. Furthermore, Dwarf leaf nodes contain aggregate values, whereas the context tree leaf nodes contain ordered sets.

B. Bloom-Based Index for the Context Tree

In order to improve the query performance of the context tree, we propose using Bloom filters [12]. A Bloom filter is a main-memory data structure that supports very efficient membership queries. When a new query for a context state is submitted by the user, instead of searching the context tree for a matching context state, the Bloom-based data structure is conducted first. Given a context state, the Bloom-based data structure provides a quick answer on whether this state exists in the tree. If the context state does not exist, then retrieving the entire context tree is avoided.

1) Bloom Filters Preliminaries: A Bloom filter is a space-efficient probabilistic data structure that is used to test whether or not an element is a member of a set. This method is used for representing a set $A = a_1, a_2, \ldots, a_l$ of l elements (also called keys) to support membership queries (*is element a in set A?*). The idea is to allocate a vector v of m bits (Fig. 11), initially all set to 0, and then choose f independent hash functions, $h_1, h_2, \ldots, h_f$, each with range 1 to m. For each element a, the bits at positions $h_1(a), h_2(a), \ldots, h_f(a)$ in v are set to 1. A particular bit might be set to 1 multiple times, but only the first change has an effect. Given a query for b we check the bits at positions $h_1(b), h_2(b), \ldots, h_f(b)$. If any of them is 0, then certainly b is not in the set A. Otherwise we conjecture that b is in the set although there is a certain probability that we are wrong. This is called a *false positive* (or a false drop) and it is the payoff for Bloom filters' compactness. The parameters k and f should be chosen such that the probability of a false positive (and hence a false hit) is acceptable. Although false positives are possible, there are no false negatives.

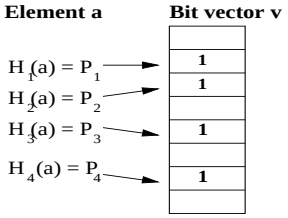

Fig. 11. A Bloom filter with f=4 hash functions.

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

In their original form, Bloom filters provided support only for simple keyword queries and not for path queries such as those representing context states. To this end, in our previous work we have introduced multi-level Bloom filters, namely $Breadth$ and $Depth$ *Bloom filters* [13], [14] to test for membership of path queries.

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

The *Breadth Bloom Filter* (BBF) for a context tree T for n context parameters is a set of n Bloom filters $\{BBF_1, BBF_2, \ldots, BBF_n\}$, where each Bloom filter, BBF_i, corresponds to an internal (i.e., non leaf) level i of the context tree, that is, there is one filter for each context parameter c_i. In each BBF_i, we insert all $keys$ that appear in cells in nodes at level i of the context tree. For example, the BBF for the context tree of Fig. 10 is a set of two Bloom filters (Fig. 12).

Depth Bloom filters provide an alternative way to summarize context trees. We use different Bloom filters to hash paths of different lengths. The *Depth Bloom Filter* (DBF) for a context tree T for n context parameters is a set of Bloom filters $\{DBF_0, DBF_1, DBF_2, \ldots, DBF_{m-1}\}$, $m \leq n$. There is one Bloom filter, denoted DBF_i, for each path of the tree with

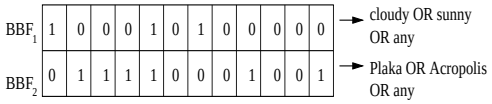

Fig. 12. The BBF for the context tree of Fig 10.

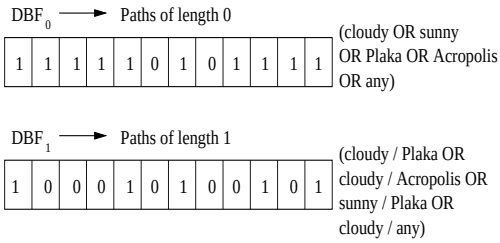

Fig. 13. The DBF for the context tree of Fig 10.

length i, that is, having $(i+1)$ nodes, where we insert all paths of length i. Note that a path with length $i < n$ corresponds to a context state in which there are values specified only for the first i parameters. Note that we insert paths as a whole, we do not hash each element of the path separately; instead, we hash their concatenation. The DBF for the context tree in Fig. 10 is a set of two Bloom filters (Fig. 13).

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

When comparing the two filters, DBF works better (has a smaller false positive ratio) than BBF [13]. The reason is that when using $BBFs$, a new kind of false positive appears. Consider the tree of Fig. 10 and the query: *sunny/Acropolis*. We have a match for $sunny$ at BBF_1 and for $Acropolis$ at BBF_2; thus we falsely deduce that the path exists. However, DBFs is less space efficient, since the number of paths is very large. This the reason, that we may not keep Bloom filters for paths of all lengths but instead we may keep paths up to a maximum length m.

VI. Prototype Implementation: Preference Restaurant Guide

Figure 14 depicts the overall system architecture of a preference database system. The Context-Aware Preference Database Management System (DBMS) stores both database relations and preferences that relate the context-dependent attributes of the relations with the context parameters. To process context-dependent queries, preferences are taken into account to present the results based on their preference score at the specified context state. We assume that the values of the current context state are provided as input to our system.

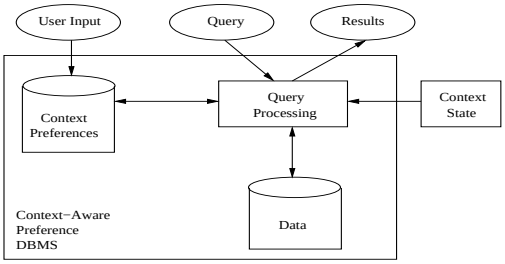

Fig. 14. Overall system architecture of a Context-Dependent Preference Database.

To demonstrate the feasibility of our approach, we have developed a prototype application based on our reference example. The application is called *Preference Restaurant Guide* and maintains information about restaurants and users. Its schema is the one depicted in Fig. 6. We consider two context parameters as relevant: $location$ and $weather$. The prototype application is build on top of Oracle 8i, using Borland JBuilder 7. The prototype implements all modules of our approach except of the context tree.

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

Besides user registration, the other part of the application includes query processing. Query processing runs in two modes: context-aware and non context-aware. In the non context-aware mode, preferences are ignored. In the context-aware mode, the user specifies the values of the context parameters she is interested in, and the results are sorted according to the user's preference in the specified context state. When no values are specified for the context parameters, the default context state that corresponds to the current context state is used. In addition, a user may use an OLAP operator to execute a *roll-up* or a *drill-down* to the results of a query. For example, suppose a result that contains restaurants located in the region of $Acropolis$. A single $roll-up$ provides restaurants in the city of $Athens$.

Suppose that user $Mary$ is at $Acropolis$ and the weather is $sunny$, i.e., the current context state is CS(current) = $\{Acropolis, sunny\}$. $Mary$ would like to know the best restaurants, according to her preferences, that are located at $Acropolis$, independently of the weather. This is an example of a query involving only basic preferences. The result of this query is depicted in Fig. 15 (left). In Fig. 15 (center) the results of a query involving aggregate preferences in the current context are depicted. If $Mary$ chooses to use an OLAP operator to execute a $roll-up$ to the results of the query, the application returns the restaurants that are located in the city of $Athens$. The results of this operation are shown Fig. 15 (right).

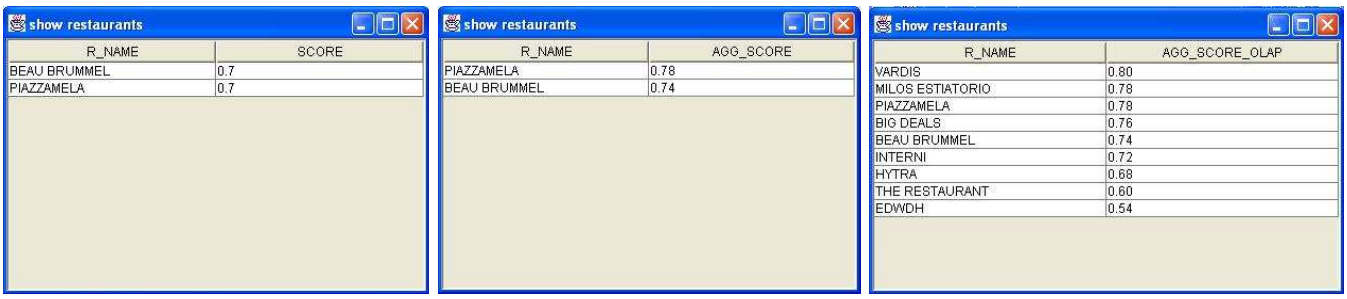

Fig. 15. The results of a query involving a basic preference on $location$ (left), on both $location$ and $weather$ (center), and a $roll-up$ to include results in $Athens$ (right).

VII. Related Work

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 a rather new research topic.

A. Context and Queries

In the context-aware querying processing framework of [15], there is no notion of preferences, instead context attributes are treated as normal attributes of relations. Query processing is divided into three-phases: query pre-processing, query execution and query post-processing. Query pre-processing is performed in two steps: a query refinement and a context binding step. The goal of the query refinement step is to further constraint the query condition by means of different contextual information. Context binding instantiates the contextual attributes involved in the refined query with exact values. After query execution, at the query post-processing phase, the results are sorted. External services may then be invoked for the delivering of results to the users. Five context-aware strategies are defined. Strategy 1 refers to queries that consider the current value of context as their reference point, for example such queries include looking for the closest restaurant, the next flight, the shortest route. To implement them, the contextual attributes are bound to their current values. Strategy 2 includes queries that access facts about the past (i.e., history data) which are recalled based on the relevant context. In this case, archived data are linked based on their common contextual attributes. Strategy 3 considers context as an additional constraint to the query. A given query is refined to include relevant constraint rules. Strategy 4 reduces the result set by ordering the produced results based on the user profile. This is achieved by using an associated sorting rule. Strategy 5 considers the delivery and presentation of results to the user by observing related delivery rules. This framework is orthogonal to our approach and a potential extension of our work includes enriching our model with constraints involving context attributes.

The Context Relational Model (CR) introduced in [16] 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.

B. 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. This is achieved by defining preferences based on context, so that under a specific context a tuple is preferred over another. 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 numerical score with every tuple of the query answer. In our work, we have adapted the general quantitative framework of [7], since it is more easy for users to employ. In this framework, a preference is expressed by the user for an entity. Entities are described by record types which are sets of named fields, where each field can take values from a certain type. The * symbol is used to match any elements of that type. Preferences are expressed as functions that map entities of a given record type to a numerical score. A set of preferences can be combined using a generic combine operator which is instantiated with a value function. For example, the preference of a user for restaurants can be expressed as preference(cuisine), with values preference(Chinese) = 0.1, preference(Greek) = 0.8 and preference(other) = 0.1.

In the quantitative framework of [17], 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. The degree of interest expresses the interest of a person to include the associated condition into the qualification of a given query. Specific rules are specified for deriving preference of complex queries by building on stored atomic ones. The results of a query are ranked based on the estimated degree of interest in the combination of preferences they satisfy. Our approach can be generalized for this framework as well, either by including contextual parameters in the atomic query elements or by making the degree of interest for each atomic query element depend on context.

There is also some similarity with the work done in the context of the PREFER system [18] for processing *ranked*

queries that is, queries that return the top objects of a database according to a preference function. The focus of this work is on a different topic: how to answer ranked queries using materialized ranked views.

In the *qualitative approach* (for example, [19]), the preferences between the tuples in the answer to a query are specified directly, typically using binary preference relations. For example, one may express that $restaurant_1$ is preferred from $restaurant_2$ if their opening hours are the same and its price is lower. This framework can also be readily extended to include context. For instance, one may express that $restaurant_1$ is preferred from $restaurant_2$ if their opening hours are the same, its price is lower and it is closer to the current user's location. A logical qualitative framework is presented in [19] for formulating preferences as *preference formulas*. The preference formula is a first-order formula defining a preference relation between two tuples.

Both the quantitative and the qualitative approached can be integrated with query processing. Relevant in this respect is research on top-k matching results and on skylines. In *top-k* queries [20], users specify target values for certain attributes, without requiring exact matches to these values in return. Instead, the result to such queries is typically a rank of the "top-k" tuples that best match the given attribute values. The *skyline* [21] is defined as those tuples of a relation that are not dominated by any other tuple. A tuple dominates another tuple if it is as good or better in all dimensions and better in at least one dimension.

Finally, the work in [22] focuses on inductively constructing complex preferences by means of various preference constructors.

C. Storing Context

An important issue is what is an appropriate model for storing context. Besides storing the current context for building context-aware systems and applications, there is growing effort to extract interesting knowledge (such rules, regularities, constraints, patterns) from large collections of context data. Storing context data using data cubes, called context cubes, is proposed in [9] for developing context-aware applications that use archive sensor data. The context cube provides a multidimensional model of context data where each dimension presents a context dimension of interest. The context cube also provides a number of tools for accessing, interpreting and aggregating context data by using concept relationships defined within the real context of the application. In this work, data cubes are used to store historical context data and to extract interesting knowledge from collections of context data. Also, a cube can be used to create new context from analysis of the existing data. In our work, we use data cubes for storing context-dependent preferences and answering related queries.

Besides queries, context parameters, such as $location$ and $time$, have been used to determine which data to replicate in mobile information systems [23]. In this approach, context parameters are called $extensions$. Finally, context has been used in the area of multidatabase systems to resolve semantic differences, e.g., [24], [25], [26] and as a general mechanism for partitioning information bases [27].

VIII. Summary

The use of context is important in many applications such as in pervasive computing where it is important that users receive only relevant information. In this paper, we consider integrating context with query processing, so that when a user poses a query in a database, the result depends on context. In particular, each user indicates preferences on specific attribute values of a relation. Such preferences depend on context. We store preferences in data cubes and show how OLAP techniques can be used to compute context-aware queries, that is queries whose results depend on context. In order to improve the performance of our system, we introduce a hierarchical data structure, called context tree. We use this tree to index results from previous queries. Finally, we demonstrate the feasibility of our approach through a prototype application regarding a context-aware preference restaurant guide.

Acknowledgment

This work was partially funded by the General Secretariat of Research and Technology of Greece through a grant for supporting bilateral research cooperations between Greece and Australia (DECA).

References

[1] A. K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing*, **5**(1): 4–7, 2001.

[2] D. Salber, A. K. Dey, G. D. Abowd. The Context Toolkit: Aiding the Development of Context-Enabled Applications. *CHI Conference on Human Factors in Computing Systems*, 434–441, 1999.

[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] E. F. Codd. A relational model of data for large shared data banks. *Communications of ACM*, **13**(6): 377–387, 1970.

[5] P. Vassiliadis and S. Skiadopoulos. Modelling and Optimization Issues for Multidimensional Databases. *International Conference on Advanced Information Systems Engineering*, 482–497, 2000.

[6] P. Sistla, O. Wolfson, S. Chamberlain and S. Dao. Modeling and Querying Moving Objects. *International Conference on Data Engineering*, 422–432, 1997.

[7] R. Agrawal and E. L. Wimmers. A Framework for Expressing and Combining Preferences. *ACM SIGMOD International Conference on Management of Data*, 297–306, 2000.

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

[9] L. Harvel, L. Liu, G. D. Abowd, Y-X. Lim, C. Scheibe and C. Chathamr. Flexible and Effective Manipulation of Sensed Context. *International Conference on Pervasive Computing*, 51–68, 2004.

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

[11] Y. Sismanis, A. Deligiannakis, N. Roussopoulos and Y. Kotidis. Dwarf: Shrinking the PetaCube. *ACM SIGMOD International Conference on Management of Data*, 2002.

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

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

[14] G. Koloniari and E. Pitoura. Filters for XML-based Service. *Discovery in Pervasive Computing*, **47**(4): 461–474, 2004.

[15] L. Feng, P. M. G. Apers and W. Jonker. Towards Context-Aware Data Management for Ambient Intelligence. *International Conference on Database and Expert Systems Applications*, 2004.

[16] Y. Roussos, Y. Stavrakas and V. Pavlaki. Towards a Context-Aware Relational Model. *International Workshop on Context Representation and Reasoning*, 2005.
[17] G. Koutrika and Y. Ioannidis. Personalization of Queries in Database Systems. *International Conference on Data Engineering*, 2004.
[18] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *Very Large Data Bases*, **13**(1): 49–70, 2004.
[19] J. Chomicki. Preference Formulas in Relational Queries. *ACM Transactions on Database Systems*, **28**(4): 427–466, 2003.
[20] S. Chaudhuri and L. Gravano. Evaluating Top-k Selection Queries. *International Conference on Very Large Data Bases*, 397–410, 1999.
[21] S. Bfrzsfnyi, D. Kossmann and K. Stocker. The Skyline Operator. *International Conference on Data Engineering*, 421–430, 2001.
[22] W. Kiesling. Foundations of Preferences in Database Systems. *International Conference on Very Large Data Bases*, 311–322, 2002.
[23] H. Höpfner and K.-U. Sattler. Semantic Replication in Mobile Federated Information Systems. *International Workshop on Engineering Federated Information Systems*, 36–41, 2003.
[24] V. Kashyap and A. Sheth. So Far (Schematically) yet So Near (Semantically). *IFIP TC2/WG2.6 Conference on Semantics of Interoperable Database Systems*, 283–312, 1992.
[25] A. Ouksel and C. Naiman. Coordinating Context Building in Heterogeneous Information Systems. *Journal of Intelligent Information Systems*, **3**(1): 151–183, 1994.
[26] E. Sciore, M. Siegel and A. Rosenthal. Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems. *ACM Transactions on Database Systems*, **19**(2): 254–290, 1994.
[27] J. Mylopoulos and R. Motschnig-Pitrik. Partitioning Information Bases with Contexts. *International Conference on Cooperative Information Systems*, 44–54, 1995.

Kostas Stefanidis is a PhD candidate in the Department of Computer Science at the University of Ioannina, Greece. He received his MSc degree in 2005 and his BSc degree in 2003 from the University of Ioannina, both in Computer Science. From 2002 until now, he is a member of the Distributed Management of Data (DMOD) Laboratory (http://www.dmod.cs.uoi.gr/). His research interests include Database and Distributed Systems.

Evaggelia Pitoura received her B.Sc. from the Department of Computer Science and Engineering of the University of Patras, Greece in 1990 and her M.Sc. and Ph.D. in computer science from Purdue University in 1993 and 1995, respectively. Since September 1995, she is on the faculty of the Department of Computer Science of the University of Ioannina, Greece. She is also a member of the Distributed Management of Data (DMOD) Laboratory (http://www.dmod.cs.uoi.gr/). Her main research interests are in data management for mobile and peer-to-peer computing. Her publications include several articles in international journals and conferences and a book on mobile computing. She received the best paper award in the IEEE ICDE 1999. She serves regularly in the program committees of most of the top conferences in distributed systems and data management.

Panos Vassiliadis received his PhD from the National Technical University of Athens in 2000. He joined the Department of Computer Science of the University of Ioannina as a lecturer in 2002. Currently, Dr. Vassiliadis is also a member of the Distributed Management of Data (DMOD) Laboratory (http://www.dmod.cs.uoi.gr/). His research interests include data warehousing, web services and database design and modeling. Dr. Vassiliadis has published more than 25 papers in refereed journals and international conferences in the above areas. More information is available at http://www.cs.uoi.gr/˜pvassil.