FOOD Index: A Multidimensional Index Structure for Similarity-Based Fuzzy Object Oriented Database Models

Adnan Yazici, Senior Member, IEEE, Cagri Ince, and Murat Koyuncu

Abstract—A fuzzy object-oriented data model is a fuzzy logic-based extension to an object-oriented database model that permits uncertain data to be explicitly represented. The fuzzy object-oriented database (FOOD) model is one of the proposed models in the literature to handle uncertainty in object-oriented databases. Several kinds of fuzziness are dealt with in the FOOD model, including fuzziness at attribute level and between object and class and between class and superclass relations. The traditional index structures do not allow efficient access to both crisp and fuzzy objects for fuzzy object-oriented databases since they are not efficient enough in processing both crisp and fuzzy queries. In this study, we propose a new index structure, namely a FOOD index (FI), to deal with different kinds of fuzziness in fuzzy object-oriented databases and to support multidimensional indexing. In this paper, we describe this proposed index structure and show how it supports various types of flexible queries, and evaluate its performance for exact, range, and fuzzy queries.

Index Terms—Flexible querying, fuzzy indexing, fuzzy set theory, object-oriented databases (OODBs), uncertainty.

I. INTRODUCTION

Among enhanced database models, object-oriented database models have attracted a lot of interest from researchers because of the introduction of the notion of nested objects and inheritance. Despite the representational power of the object-oriented paradigm, very few studies on fuzzy data representation in object-oriented databases (OODBs) exist. Fuzzy index structure is an issue to be investigated for OODBs, because many complex database applications require manipulation of fuzzy and imprecise data [16]. In such applications, data may not be precise and users of a database may be interested in asking imprecise (or fuzzy) queries. For example, an eyewitness may see the person who committed a crime but be unable to describe the person accurately. In other words, the description of the characteristics of the suspect is fuzzy. A fuzzy-logic-based extension to the data model is a possible solution that allows imprecise or fuzzy data to be explicitly represented [11], [12]. One example of fuzzy object-oriented database (FOOD) modeling is the similarity-based fuzzy object-oriented data model, known as the FOOD model [16], [23], [24]. The FOOD model allows an accurate representation of various types of uncertainty in the OODB model enhanced with fuzzy concepts and is the reference data model throughout this paper.

The indexing of FOODs [18], [21], [22], has not received much attention yet. Index structures allow fast access to data by content. The current crisp index structures [3], [5], [14], [15] developed for OODBs are inappropriate to represent and efficiently access fuzzy data for the FOOD model. Fuzzy querying allows one to express vague predicates represented by fuzzy sets. Conventional index structures cannot be used directly since fuzzy predicates may not refer to the entry values of the index. Therefore, an efficient indexing mechanism for the FOOD model is needed to allow fast access to the objects with crisp or fuzzy values. In order to support exact, range, and fuzzy queries efficiently, a multidimensional index structure that can use both crisp and fuzzy attributes as organizing attributes of objects should be used. Therefore, in this study, we propose a new index structure, a FOOD index (FI), dealing with different kinds of fuzziness in the FOOD model and supporting multidimensional indexing. The FI structure handles various types of flexible queries including crisp, range, and fuzzy queries.

A number of indexing techniques for object-oriented data models exists, such as path index [5], nested inherited index [3], [15], enhanced nested inherited index [3], [15], and others [3], [6], [14], [15]. However, none of these supports fuzzy querying and multidimensionality. The indexing techniques defined in the framework of object-oriented data models are either structural [3], [6], [14], [15], [20] or behavioral [7]. Structural indexing is based on object attributes and can be classified into techniques supporting nested predicates, such as the ones presented in [6], [14], and [20], and techniques supporting queries issued against an inheritance hierarchy [15]. On the other hand, behavioral indexing aims at efficient execution of queries containing method invocations. It is based on precomputing or caching the results of a method and storing them in an index. The major difficulty in this approach is the detection of changes invalidating the results of a method.

On the other hand, very few indexing methods have been proposed for fuzzy relational database models [8], [13], [23], [25]. The fuzzy indexing structure proposed in [8] uses one index per fuzzy predicate tied to an attribute. This indexing structure only deals with homogeneous domains and assumes that the underlying relations are crisp. Another study on fuzzy indexing introduces a fuzzy index structure, but it is developed for fuzzy relational database models [23], [25]. One other study...
that is presented in [13] includes several methods to index fuzzy sets in databases to improve query evaluation performance. The access methods in [13] are based on superimposed coding or rely on inverted files.

The main contribution of this paper is the introduction of a new multidimensional indexing technique (namely the FI) for FOOD systems. The path index and the other existing OODB index structures are not multidimensional. Although some multidimensional index structures such as k-d-tree [2], G-tree [17], [19], UB-tree [1], etc., exist, they cannot be used for indexing FOOD systems as they cannot handle fuzzy data and fuzzy querying. Current object-oriented index structures [3]–[7], [14], [15], [20] are developed for crisp OODBs and cannot be used for fuzzy queries, whereas the FI can efficiently handle both crisp and fuzzy data and queries. We have devised a method to create common bit strings (BSs) for crisp and fuzzy attribute values. By using these BSs, exact, range, and fuzzy queries are dealt with. We have also improved the performance of the queries that the path index supports by reorganizing the directory structure used in the path instantiation node (PIN). Finally, we have implemented the FI structure and compared its performance results with that of the path index. The reason for using the path index for comparison is that it is one of the best known index structures for OODBs in terms of retrieval efficiency [4], and we show that the FI performs better than the path index for various types of queries. This paper is an extension of the study included in [26]. Besides the extension done in each section of this paper, we also introduce the extension of path index, insertion, deletion, and retrieval algorithms of the FI and some other examples, along with the detailed performance tests, including multidimensionality.

The remainder of the paper is organized as follows. Section II includes some necessary background about the FI. In Section III, the FI structure is described in detail. The performance evaluation of the FI structure for various types of queries, such as crisp exact match, crisp range, and fuzzy queries, are presented in Section IV. Finally, Section V concludes the paper.

II. BACKGROUND

This section provides the background required to better understand the construction of the FI. Preliminary definitions and notations are given in Section II-A. The FOOD model for which the FI is developed is described in Section II-B. In Section II-C, the path index and brief information on the related topics are introduced. For details, see the references [12], [16], [23], and [24] for the FOOD model and see [5] and [6] for the path index.

A. Preliminary Definitions and Notations

We assume objects to be organized in classes and classes to be organized in an inheritance as well as aggregation hierarchy. For the sake of simplicity, we also assume that hierarchies do not contain cycles. A class is defined by specifying its name, its attributes, its methods and its super classes with a degree of membership. In the FOOD model, an attribute is defined by specifying its name, range, and relevance.

A path is defined as \( C_1, A_1, A_2, \ldots, A_n \) \((n \geq 1)\) [3], [4] where:
1) \( C_1 \) is a class in the class hierarchy;
2) \( A_1 \) is an attribute of class \( C_1 \);
3) \( A_i \) is an attribute of a class \( C_i \) in the class hierarchy and \( 1 < i \leq n \).

A path is a sequence of class and attributes in an aggregation hierarchy. For instance, when the given class has an attribute with a domain of the second class of the path (the attribute of the class has an object from second class as its value), the second class has an attribute with a domain of the third class of the path, and so forth. Fig. 1 shows an example database schema, which will be used as a reference database schema throughout the paper. The solid arrows representing the aggregation hierarchies and multivalued attributes are highlighted with “•”. In the aggregation hierarchies in this figure, a library has members, a member may borrow books from the library, a book has authors, and an author has some fuzzy attributes such as age, height, and subject area. The dashed arrows show the inheritance hierarchy and the numbers on the arrows are the degrees of membership of a class to its superclass(es). For example, the class Student is a subclass of the class Educated with a membership degree (MD) 0.9, while at the same time, it is a subclass of the class Highly_Educated with an MD 0.6.

A sample index path for the database schema is \( P: \text{Library}.\text{Members}.\text{Books}.\text{Author}.\text{Age} \), and for this path, we have the following definitions and notations.

1) The length of the path, denoted by \( \text{len}(P) \), is the number of classes on the path. \( \text{len}(P) = 4 \).
2) The set of classes on the path is denoted by \( \text{class}(P) \).
\( \text{class}(P) = \{ \text{Library}, \text{Member}, \text{Book}, \text{Author} \} \)
3) The classes on the path and all of their subclasses form the scope of the path are denoted by \( \text{scope}(P) \).
\( \text{scope}(P) = \{ \text{Library, Member, Highly_Educated, Educated, Lecturer, Assistant, Student, Book, Science, Engineering, Computer_Science, Computer_Engineering, Algorithms, Robotics, Database, Author, Native_Author, Foreign_Author} \} \).
4) The position of a class on the path, denoted by \( \text{pos}(\text{class}) \), is the number denoting its place on the path, starting from the left hand side. \( \text{pos}(\text{Library}) = 1 \), \( \text{pos}(\text{Educated}) = 2 \), \( \text{pos}(\text{Robotics}) = 3 \), \( \text{pos}(\text{Native_Author}) = 4 \).
5) A fuzzy object of any class is denoted by the abbreviation of that class, followed by a class-wide unique identifier within square brackets. For example, \( \text{LB}[1] \) is the first object of the class Library (\( \text{LB}[1] \) is the object ID). Some sample fuzzy objects and their attribute values for the database schema given in Fig. 1 are listed in Fig. 2.

The graphical representations of the relations among the sample objects are shown in Fig. 3. We can see from the figure that \( \text{LB}[1] \) has five members (i.e., \( \text{LC}[1], \text{AS}[1], \text{AS}[4], \text{ST}[1], \) and \( \text{ST}[2] \)). Similarly, \( \text{LC}[1] \) has borrowed two books (i.e., \( \text{AB}[2] \) and \( \text{RB}[2] \)) and \( \text{AB}[2] \) is written by two authors (i.e., \( \text{NA}[3] \) and \( \text{FA}[3] \)). A path instantiation (PI) is a sequence of objects found by instantiating a path. Considering the objects in Fig. 3, \( \text{LB}[1].\text{AS}[4].DB[3].FA[5].\text{Old}, \text{AB}[1].\text{NA}[1].\text{Old}, \) and \( \text{ST}[2].\text{DB}[1].\text{FA}[4].45 \) are examples of PIs for the path \( P \).
B. Similarity-Based FOOD Model

The basis of the FOOD model [12], [16], [23], [24] is the replacement of equality with a similarity relation. Similarity matrices (as shown in Table I) are used to represent similarity relations between values of fuzzy attributes. The domain, dom, is the set of values that the attribute may take, irrespective of the class that it falls into. The range definition for attribute \( a_i \) of class \( C \) is represented by the notation \( \text{rng}_C(a_i) \), where \( a_i \in \text{Attr}(C) \). The range of an attribute \( a_i \), \( \text{rng}_C(a_i) \), is the set of allowed values that a member of a class \( C \) may take for an attribute. In general, for each attribute of a class, \( \text{rng} \subseteq \text{dom} \). For example, if a fuzzy domain for age attribute is \( \text{dom} = \{ \text{young}, \text{middle-aged}, \text{old} \} \), then a range will be a subset of this domain definition, such as \( \text{rng} = \{ \text{young}, \text{middle-aged} \} \). \text{Attr}(C) refers to the attributes of class \( C \). A relevance weight \( \text{RLV}(a_i,C) \) is assigned to each attribute to represent the significance of the range definition of that attribute \( (a_i) \) on the class \( (C) \) definition.

![Fig. 1. Example database schema.](image)

![Fig. 2. Sample fuzzy objects for the database schema in Fig. 1.](image)

**TABLE I**

| SIMILARITY MATRIX FOR FUZZY-VALUED ATTRIBUTE COLOR |
|----------------------------------|-------------|-------------|-------------|
| red                | orange      | white       |
| red                | 1           | 0.6         | 0           |
| orange             | 0.6         | 1           | 0.1         |
| white              | 0           | 0.1         | 1           |

Similar objects are grouped together to form a class and fuzzy relations are taken into consideration at class–object and class–subclass levels. Based on the relevance and ranges of attribute values, the membership of object \( o_j \) in \( C \) is formulated as

\[
\mu_C(o_j) = \frac{\sum \text{INC}(\text{rng}_C(a_i)/o_j(a_i)) \times \text{RLV}(a_i,C)}{\sum \text{RLV}(a_i,C)} \tag{1}
\]

where \( \text{RLV}(a_i,C) \) indicates the relevance of the attribute \( a_i \) to class \( C \), and \( \text{INC}(\text{rng}_C(a_i)/o_j(a_i)) \) denotes the degree of inclusion of the attribute values \( a_i \) of \( o_j \) in the formal range.
of \(a_i\) in class \(C\) \([\text{rng}_C(a_i)]\). The inclusion degree is calculated by using the similarity of \(o_j(a_i)\) to \(\text{rng}_C(a_i)\) and denotes the strength of similarity of the object attribute values to the class range values. Note that the membership of a subclass (object) in a superclass (class) may be prespecified by the designer. The FOOD model is a practical approach for the representation of imprecise data through the use of similarity matrices.

1) Attribute Definitions: A fuzzy object can have attributes in the form of a set of values from a fuzzy domain. The similarities between the elements in that fuzzy domain are represented using a similarity matrix. For instance, consider a class, BOOK, with an attribute defined in the domain, Color = \{red, orange, white\}, with the similarity matrix shown in Table I. According to the table, the degree of similarity between the elements “red” and “orange” is 0.6.

Every class has a range definition for each of the fuzzy attributes with the corresponding relevance values indicating the importance of that attribute in the definition of that class. Range values show ideal values of a class. However, an attribute of a class can take any value from a domain without considering the range values. Object/class MDs are calculated from the similarity of object values to the range values using formula (1). Similarly, class/superclass MDs are calculated from the similarity of range values of two classes. In this model, semantics is associated with range definitions to permit a more precise definition of a class. When an attribute is multivalued, semantics defines the relationships among these values. There are three semantics used in the FOOD model: AND, OR, and XOR. AND semantics requires objects to have all the values given in the range definition, OR semantics requires objects to have at least one of the values given in the range definition, and finally, XOR semantics requires objects to have exclusively one of the values given in the range definition. The details of the attribute representations and the other features of the FOOD model are described in [16], [23], and [24].

2) Object/Class Relations: The object/class level denotes the membership of an object to a class. The main feature that distinguishes fuzzy classes from crisp classes is that the boundaries of fuzzy classes are imprecise. The imprecision of the attribute values causes imprecision in the class boundaries. Some objects are full members of a fuzzy class with an MD 1, but some objects may be members of this class with a degree between 0 and 1. The MD of an object of a class is calculated using the similarities between the attribute values, the class range values, and the relevance of fuzzy attributes. The closer the attribute values to the range, the higher the MD of the object to the class.

3) Class/Superclass Relations: The relationship between a class and its superclass answers the following query: “To what extent does the class belong to its superclass?” In the FOOD model, the class/superclass relation may be fuzzy. Therefore, a class may be connected to its superclass(es) with an MD. Since the FOOD model is similarity-based, MD of a class to its superclass is calculated taking into account the similarity of the attribute range definitions. For details of the FOOD model with examples of the computation of class/superclass MDs, see [23] and [24].

C. Path Index

A path index [5], [6] is a data structure for indexing OODBs along with both aggregation and inheritance hierarchies. It associates an object \(O\) at the end of a path with all instantiations ending with \(O\), and therefore, it can be used to solve nested predicates against all classes along the path.

A path index, defined on a path \(P = C_1.A_1.A_2.A_3\cdots A_n\), is implemented by a single \(B^+\)-tree. The leaf nodes and the nonleaf nodes of such a \(B^+\)-tree have different formats. A nonleaf node consists of \(f\) records structured as \(<\text{key length}, \text{key value}, \#\text{ of PIs, the list of PIs}>\), where the pointer contains the physical address of the next-level node and \(f\) is the order of the \(B^+\)-tree. When the number of records in a leaf node is less than the page size (maximum record number), the leaf node has the structure \(<\text{record length}, \text{key length}, \text{key value, \# of PIs, the list of PIs}>\). Otherwise, a directory is formed in the record as in Fig. 4 and the PIs are ordered. In such a directory, for each page occupied by the list of instantiations, we store the pair \((PI, page\ address)\), where \(PI\) is the last instantiation stored in that page. In Fig. 4, the PIs are indexed as \(PI_1, PI_2, \ldots, PI_k\), if they fit in a page. When PIs do not fit in a page, a directory is created and PIs are indexed as \(PI_{11}, PI_{12}, \ldots, PI_{1M}\) to show the new level, where the first number shows the directory number and the second number shows the order of the PIs. That is, \(PI_{11}\) is the first PI in the first directory. PIs are implemented as arrays of object identifiers (OIDs) of the objects on the path. Each object has a unique OID. The length of the array is the maximum length of the index path.

III. FI Structure

The FI is a multidimensional indexing technique, which means one or more than one organizing attribute(s) belonging to the same class (such as the age and height attributes of a
A. Adaptation of the Path Index Structure

The path index structure is utilized to handle both aggregation and inheritance hierarchies in the FOOD model. Search key values can be fuzzy or crisp. Therefore, we convert fuzzy and crisp values into a common base to form a search key. Our approach to this problem is to map fuzzy and crisp values to a BS formed by 0’s and 1’s to provide a common base for the two types of values. The mapping is explained in Section III-B. The FI utilizes a structure similar to the $B^+$-tree to index both fuzzy and crisp values, forming a BS as the search key.

The whole FI structure is shown in Fig. 5. The tree structure is used to reach the data buckets starting from the root node of the FI. In the data buckets, there are data bucket records for each different key value indexed. Using the key value, we can access the data bucket, and thus, the data bucket records related to that key value. Each data bucket record has a pointer to the PIN that stores all the PIs related to the key value of that data bucket record.

Fig. 6 shows the structure of the nonleaf and leaf nodes of the FI. Each node has a number of node records determined by the node factor (NF). Each node record has a BS as a key and a pointer to the next level. In a nonleaf node, the next level is again a node while the next level in a leaf node is a data bucket. Moreover, a node has pointers to the previous and next sibling nodes.

Fig. 7 shows the structure of a FI data bucket. A data bucket contains a number of data bucket records determined by the data bucket factor (DBF). Each data bucket record has a BS as a key and a pointer to its PIN.

Each PIN has a number of PIs determined by the path instantiation node factor (PINF). Within the PIN, all the PIs that
are related to the key value of the data bucket record are stored. While searching, when a PIN is accessed, the OIDs in the position of the target class within its PIs are determined. The object referred to by these OIDs form the result set of the query.

As in the path index, the FI stores the PIs related with the same key value in a single PIN if they fit into a page as shown in Fig. 8. As explained in Section II-C, PIs are implemented as arrays of OIDs of the objects on the path.

Note that OIDs uniquely identify the objects in a FOOD database. However, OIDs do not include the class identification number since in the FOOD model, the class that an object belongs to can change when the attribute values of the object change because of fuzziness. The class information of the object is dynamically stored within the object itself.

One of the differences between the path index and the FI is the organization of the PIN in cases where the instantiations of the paths do not fit into a page. In the path index, PIs are ordered and a directory is kept at the beginning of the record, which forms links to the lower level pages. In this directory, the last instantiation and the address of that page are stored for each page occupied, as described in Section II-C. A possible worst case for the path index is shown in Fig. 9(a). The assumption is that we search for the objects whose positions in the aggregation hierarchy are 4 and the only PI with length 4 is \( P_{N_m} \), which is at the end of the directory list. The length of other PIs are shorter than 4. For the path index case, the complete directory list must be traversed linearly until \( P_{N_m} \) is reached.

In the FI, we use a new directory structure to handle such worst cases. In the FI, the PIs are clustered according to their lengths, as shown in Fig. 9(b). PIs whose lengths are the same are kept in the same lower level page if possible; otherwise, a linked list of lower level pages is formed and PIs are stored on the lower level pages in a disordered fashion, since all of them are within the scope of the result. Similar to the path index, a directory is kept at the beginning of the record to access the clusters of different lengths. For each cluster, a key to the length of the PIs in that cluster and a pointer to the beginning of the linked list of the lower level pages are stored in the directory. The keys in the directory are ordered according to the length, from lower to higher values. For the same worst case, we can directly access the related pages in the FI to retrieve the PIs with a length of 4 without any need to traverse all the directories of PIs of a length less than 4.

Reorganization of the PIN makes the FI perform better than the path index for crisp and range type queries. The reason is that, when the PIN is accessed, the focus is on the PIs that contain the object for the target class of the query. For example, considering the database schema in Fig. 1, if we want the names of the libraries whose nested attribute \( \text{age} \) has a value of 45, then we are only interested in PIs whose length is 4, but not less than that. In such a case, using the directory in the FI, we directly access the PIs whose length is 4 (meaning they contain a library object) instead of accessing many pages with PIs containing no library objects at all in the path index. In the case of insertion and deletion operations in the database, again the FI has a better performance than the path index. The reason is that the FI does not store the PIs in a sorted fashion.

The differences between the FI and the path index structures can be stated briefly as follows: First, the FI uses BSs to access the leaf nodes whereas the path index uses key values. Second, the FI structure uses a different directory structure when PIs do not fit in a page. Third, the FI is a multidimensional index structure whereas the path index is not. Finally, the FI is a fuzzy index structure supporting fuzzy and crisp values indexing together, whereas the path index cannot be used to index fuzzy values, but only crisp values.

B. Representation of Crisp and Fuzzy Attribute Values

A fuzzy attribute can contain either a fuzzy value or a crisp value. For example, if the fuzzy attribute is \( \text{age} \), it can be a crisp value like “65” or a fuzzy term like “old”. Thus, we need to convert the crisp and fuzzy values into a single common base in order to index them in the same structure. As mentioned before, we use a multidimensional data structure with the BSs to determine which path to follow in order to reach the leaf nodes. As a result, the crisp and fuzzy values are represented in a BS and we utilize the \( B^+ \)-tree like structure to reach the PIs related to the key values sought. The search is guided by using the composite BSs, and therefore, we concentrate on how to construct these BSs.

Organizing attributes, which are used together for indexing, are a subset of all attributes of a class. For example, the attributes \( \text{age} \) and \( \text{height} \) of a class \( \text{person} \) can be used to organize the class \( \text{person} \) by indexing them together. The constructed BSs for different organizing attribute values must be different from each other so that they can be differentiated while searching in the index structure. In order to achieve this, the index space that includes all the possible BSs for any organizing attribute value is partitioned according to the characteristics of the organizing attribute values. The FI uses the logical partitioning shown in Fig. 10, which gives an example of the fuzzy-valued attribute \( \text{Age} \) for a 1-D case.
**Fig. 9.** PIN and directory structure when PIs do not fit into a page. (a) Directory structure used in the path index as a worst-case scenario. (b) Directory structure used in the FI for the same case in Fig. 9(a).

**Fig. 10.** Example of index space partition.

The BS structure of a fuzzy-valued attribute is formed as follows:

\[
< T_1 \cdots T_n R_1 R_2 M_1 \cdots M_m V_1 \cdots V_S >.
\]

**T_1 \cdots T_n:** The index space is firstly partitioned according to the fuzzy terms of the organizing attribute. The BS \( T_1 \cdots T_n \) identifies the fuzzy term, i.e., “001”: young, “010”: middle-aged, “100”: old, “011”: young or middle-aged, etc. If an attribute value to be indexed is fuzzy, it is represented in binary using this encoding method. On the other hand, if an attribute has a crisp value, the corresponding fuzzy term is determined using membership functions. There is a membership function for each fuzzy term in a domain as shown in Fig. 11. In BSs, we keep the fuzzy term (to which the crisp value mostly belongs) in addition to the crisp value itself. For example, among membership functions of the fuzzy-valued attribute *age* shown in Fig. 11, crisp value 40 is represented by the fuzzy term *middle-aged* in the \( T_1 \cdots T_n \) part of the BS.

**R_1 R_2:** The Index space for each fuzzy term is repartitioned according to the region number that the attribute value belongs to. The BS \( R_1 R_2 \) represents the region number that the MD belongs to, i.e., “00”: Region_0, “01”: Region_1, “10”: Region_2, “11”: Fuzzy Value.

The graph of any membership function, as shown in Fig. 11 for *middle-aged*, has three major regions (parts): the region in which the MD increases from 0 (zero) to 1 (one) is called Region_0 and it is identified by the BS “00”. In Region_1, which is identified by the BS “01”, the MD stays at 1 (one). Region_2 covers the region where the MD decreases from 1 (one) to 0.
(zero), and is identified by the BS “10”. Note that region numbers are only valid for crisp values since a fuzzy attribute value may belong to any region of the membership function. Thus, fuzzy attribute values are represented by the BS “11”. The regions are used to differentiate the crisp and fuzzy attribute values while dividing the index space into smaller pieces.

\[ M_1 \cdots M_m, \]  

The MD is considered to partition the index space for each fuzzy term and region number pair into smaller pieces. The BS \( M_1 \cdots M_m \) represents the MD of an attribute value for the fuzzy term that it mostly belongs to. The MD is represented in base 2. \( M_1 \) represents the integer part while the rest represents the decimal part of the MD, i.e., for \( \mu = 1.0, M_1 M_2 M_3 = “100” \), for \( \mu = 0.5, M_1 M_2 M_3 = “010” \), for \( \mu = 0.75, M_1 M_2 M_3 = “011” \). Note that in theory, there is no limitation to the number of bits that specify the MD. However, after some bits, the MD ceases to be useful to distinguish records. Therefore, we can limit the number of bits in the BS representing the MD.

\[ V_1 \cdots V_S; \]  

Finally, we partition the index space according to the crisp values in order to differentiate the crisp values from each other. The BS \( V_1 \cdots V_S \) represents the actual crisp value. If the value is fuzzy, 0 (zero) is stored for all digits.

For example, assume that “Ahmet” is 15 years old. Using the membership functions defined for the attribute “age,” we calculate that age 15 is “young” with an MD of 1.0. The MD is represented in base 2. \( M_1 \) represents the integer part while the rest represents the decimal part of the MD, i.e., for \( \mu = 1.0, M_1 M_2 M_3 = “100” \), and that it is in Region 1. Thus, starting from the top of the index tree in Fig. 10, we access the node young, constructing the BS as “001” for the fuzzy term young; then we access the node Region 1 adding “01” for Region 1 to the BS yielding “00101”. Since \( \mu_{young}(15) = 1.0 \), the node for the MD 1.0 is accessed next and “100000”, which represents 1.0, is added to the BS. The BS now becomes “00101100000000000”. Afterwards, accessing the corresponding crisp value node, “001111”, which represents the original crisp value 15, is added to the BS. Eventually, we end up with “00101100000000001111” as the final BS constructed for indexing.

C. Multidimensional Indexing

The number of organizing attributes determines the dimension of the index. This means that if we have two organizing attributes, a 2-D index is required. For each dimension, one BS is constructed according to the attribute value for that dimension as explained earlier. In other words, data in each dimension are handled as in a 1-D case. However, the FI can use only one BS as an input in order to reach the leaf nodes related to the keys. Hence, the BSs constructed for each dimension are combined into a single BS for a multidimensional case. In this paper, the proposed combination of BSs for multidimensional indexing is

\[ <B_{1,1} B_{2,1} \cdots B_{n,1} B_{1,2} B_{2,2} \cdots B_{n,2} \cdots B_{1,m} B_{2,m} \cdots B_{n,m}> \]

where \( B_{i,j} \) is the \( i \)th bit of the BS of a fuzzy-valued attribute that belongs to the \( i \)th dimension. More clearly, the combined BS is obtained by joining each bit of the BSs for each dimension one by one according to the order among the dimensions, i.e., the first bit of the first dimension, the first bit of the second dimension, . . . , the first bit of the \( i \)th dimension, the second bit of the first dimension, the second bit of the second dimension, . . . , the \( j \)th bit of the \( i \)th dimension. The order of the dimensions is not important as long as the same ordering is used for constructing the multidimensional BSs.

For instance, suppose that the BS representation of the attribute value “age 15” for the first dimension, age, is calculated as “001011000000001111” and the BS representation of the attribute value “medium” for the second dimension, height, is constructed as “01101100000000000”. Then, the following BS is constructed for a 2-D index

\[
\begin{align*}
001011 & 0001111 \\
010111 & 0000000 \\
001100111 & 110000000 \ 0000001010 & 1010.
\end{align*}
\]

The combined BS partitions the index space in a way similar to that illustrated in Fig. 10. The only difference is that the index space is partitioned according to the Cartesian product of the values for each dimension. For example, for age and height attributes, suppose that each dimension uses 3 bits for the fuzzy terms. Thus, the first 6 bits in the combined BS partition the index space according to the Cartesian product of the fuzzy terms of each dimension, i.e., Young–Short (000111), Young–Medium (000110), Young–Tall (010010).

D. Handling Object/Class and Class/Superclass Relations

An SQL-like query format is used in the FOOD model. The query has three main parts: a select clause, a from clause, and a where clause. The conditions defined in the where clause are called “where conditions.” For fuzzy valued conditions, the threshold value for the fuzzy term is placed next to the condition that relates to attribute level fuzziness. Threshold value represents the minimum membership level to be satisfied by the queries. There is no need for a threshold value if the condition is crisp. Another condition is the “from condition.” Because of the fuzziness in the object/class and class/superclass relations in the FOOD model, an object may not fully belong to any class, but it might partially belong to a class with a degree calculated by formula (1). Thus, we have to specify another threshold value for each class within the from clause of the query. The from clause of a query in the FOOD model gives the meaning “only deal with the objects belonging to those classes or one of their subclasses with a degree greater than or equal to the threshold levels specified in the query.” The threshold levels for each class are positioned consecutively in the from clause.

Within the select clause, the question “what information will be retrieved as a result of the query” is answered. The following example shows the retrieval of the name and surname of the member objects that belong to the class Member (see Fig. 1) with a minimum MD of 0.6, if its nested attribute Age is Old with a minimum MD of 0.8, and if its nested attribute Height is greater than 170 and the author object of those attributes is a
Using the FI, the PINs that satisfy the *where* conditions of the query are sought and the PIs of the target class or its subclasses are obtained. However, the objects belonging to the target class or one of its subclasses are only possible results of the query, since we have not yet dealt with the factors of object/class and class/superclass relations. Unlike other object-oriented database models, the objects in the FOOD model belong to classes at varying degrees. Thus, a threshold value is defined for each class specified in the *from* clause. We have to check if the PI satisfies the *from* condition as well. For the example given earlier, the path is between the member class and the native_author class (see Fig. 1). For these two classes, selected objects should have object MDs equal to or greater than the threshold levels specified in the query. For the other classes on the path (i.e., the book class in this example), we use a minimum of the threshold levels specified in the query (i.e., min(0.7, 0.6) = 0.6). Therefore, the PI, in which the objects of the book class have an MD equal to or greater than 0.6, will be considered; others will be eliminated.

In addition, since an object may not directly belong to the target class but to its subclass, its subclasses are also within our scope. Similar to the object/class relation, the MD of a class to its superclass is calculated using similarities between the attribute range values of two classes and the relevance of fuzzy attributes [23]. Since the class definitions do not change frequently, the MD of a class to its superclass(es) may be kept in a table to speed up the calculations.

Fig. 1 shows an example of class hierarchy for a class/superclass relation. The dashed arrows show the inheritance hierarchy and the numbers on them are the degrees of membership of a class to its superclass(es). The member class has two subclasses: highly_educated and educated, and the MDs for these classes are 1.0. The educated class has two subclasses, assistant and student, and the MDs for these classes are 1.0 and 0.9, respectively.

The MD of two classes, if there is no direct relation between them, is calculated by combining the MDs of the classes with their subclasses along the inheritance hierarchy to obtain a single value. In this study, we use the “maximum operator” to combine MDs. For example, using Fig. 1, the MDs between the student class and the member class are combined as follows:

\[
\mu_{\text{member}}(\text{student}) = \max(\mu_{\text{educated}}(\text{student}), \mu_{\text{member}}(\text{educated})) = \max(0.9, 1.0) = 1.0.
\]

However, there is another path that can be followed from the class student to the class member (student, highly_educated, member) and we obtain again \( \mu_{\text{member}}(\text{student}) = 1.0 \). When there is more than one path, the path that yields the highest MD is considered. In our example, both paths result in the same value, and the MD of the student class to the member class is calculated as 1.0.

In this sense, the MD of an object of the *student* class to the *member* class is calculated as follows:

\[
\mu_{\text{student}}(\text{student}) = 1
\]

\[
\mu_{\text{member}}(\text{student}) = \min(\mu_{\text{student}}(\text{student}), \mu_{\text{member}}(\text{student})) = \min(1.0, 1.0) = 1
\]

where the first value is the object MD of student1 to the class student and the second value is the MD of the class student to the class member.

After calculating the MDs of the objects, the PIs containing the objects with an MD less than the threshold value of the index class defined in the query are eliminated, since they do not satisfy the condition given in the *from* part of the query.

Next, using the rest of the PIs, the OIDs are obtained for all the objects whose (nested) attribute \( A_n \) satisfies the *where* conditions of the query and which belong to the target class or to one of its subclasses. As the last step, the attribute values of those objects defined in the *select* clause of the query are obtained and retrieved as the result of the query.

E. Insertion Algorithm

Inserting an object into the FOOD requires the reference of that object to be inserted into the FI so that it can be accessed easily. Fig. 12 shows the insertion algorithm of the FI, which is defined on the path \( P = C_1, A_1, A_2, A_3, \cdots, A_n \), and an object \( X \), which is a member of one of the classes on the path, is inserted into the database.

Example: We now construct the FI structure starting with an empty database. Initially, the root is null. The values of DBF, NF, and PINF are all equal to 3 in this example. 2-D indexing is constructed on *age* and *height*. If the BS used for indexing each dimension has a length of 32, then the combined BS has a length of 64 for 2-D indexing. In Fig. 13, we insert the object NA[1] of Fig. 3 into the index. This process takes place as follows.

1. Starting from the object NA[1], a 2-D value set \( S \) is formed with a value of \( \{(\text{old}, \text{short})\} \). The PIs ending with the 2-D value \( \{(\text{old}, \text{short})\} \) form the set \( \text{SP}(\text{old}, \text{short}) = \{\text{NA}[1]\} \).
2. A BS for the value \( \text{old} \) and a BS for the value \( \text{short} \) are constructed. The two BSs are combined bit by bit, forming a 2-D BS.
3. Using the constructed BS, the root of the FI is accessed to reach the corresponding data bucket. Since initially, the root node is null, a data bucket is created; then the data bucket record is created and inserted into the bucket. A PIN is created and the data bucket record is linked to it.

Then, NA[1] is inserted as the first PI in the created PIN. The FI structure at this point (after inserting the first object) is shown in Fig. 13.

The FI structure obtained is shown in Fig. 14 after inserting objects NA[4] and FA[1]. When we insert FA[2] into the database, a data bucket is created because of the overflow in the data bucket. The 2-D space is divided into two: the data bucket
records whose BS starts with 0 and those whose BS starts with 1. Then, a new root node is created with a two-node record where the former represents the BS starting with 0 and the latter represents the BS starting with 1. As a result, the data bucket records that already exist are redistributed to either the old or the new data bucket according to their BSs. When we insert the objects, \( AB[4] \), \( RB[4] \), \( DS[4] \), \( AS[5] \), \( ST[5] \), \( LB[4] \) into the database with the PIs \( AB[4].NA[4] \), \( AS[5].RB[4].NA[4] \), and \( LB[4].ST[5].DS[4].NA[4] \), the resulting index structure shown in Fig. 15 is obtained.

When we want to insert \( AB[5] \) with the PI \( AB[5].NA[4] \) into the database, we see that there is no empty space for the new PI. Thus, a directory page is allocated and the PIs are clustered according to their length (as shown in Fig. 16). For each cluster, a key (the length of the PIs in the cluster) and a pointer to the first page of the cluster are kept in the directory page.

**F. Deletion Algorithm**

Deleting an object from the FOOD database requires the reference of that object in the FI to be deleted. Therefore, when any object is deleted from the database, there is a forward traversal starting from the object in order to obtain the nested attribute values of the indexed attributes. Fig. 17 presents the deletion algorithm for a case where there is a FI defined on the path \( P = C_1.A_1.A_2.A_3 \cdots A_n \) and an object \( X \) which is a member of one of the classes on that path is deleted from the database.

**Example:** If we assume we want to delete the object \( AB[5] \) from the database, this deletion causes the deletion of the PI \( AB[5].NA[4] \) in the index structure given in Fig. 16. When we delete this PI from the index structure, it results in three PIs connected to the PIN directory. However, three PIs can fit in a single PIN. Therefore, a merge operation is carried out resulting in the index structure presented in Fig. 15.

**G. Retrieval Algorithm**

A general retrieval query in the FOOD model has three main parts: select, from, and where, as explained in Section III-D. The retrieval algorithm (shown in Fig. 18) takes a query as an argument and returns the values of the specified attribute(s) of the objects. The retrieval algorithm first determines the type of query (crisp exact match query, crisp range query, or fuzzy query), and then, constructs the BS list composed of the start and stop BS pairs accordingly. When a query requires retrieving objects from a certain range, the start and stop BS pair specifies the starting and ending point in the FI structure for this range.
The objects belonging to the target class and existing in the range of the start and stop BS pairs with an MD greater than the threshold specified in the from clause are collected and the values of the specified attribute(s) of the objects are brought back as the result. In this algorithm, the critical point is the construction of the BSs. Therefore, the construction of the BSs is explained depending on the query types: crisp exact match query, crisp range query, and fuzzy query.

**Crisp Exact Match Query:** Crisp exact match conditions are in the form

\[ \text{indexed attribute} = \text{crisp attribute value} \]

We compute the region and the MDs of the crisp value depending on the fuzzy terms it belongs to in the domain. The fuzzy term for which the crisp value has the highest MD among all fuzzy terms in the domain is used while constructing the start BS. Since the exact match condition defines a point in space, the stop BS is the same as the start BS.

For example, if we assume the where condition is “Age = 53” and if we use the membership functions given in Fig. 19, then we compute the MDs and region numbers of the attribute value 53 for the fuzzy terms as follows:

\[ \mu_{\text{young}}(53) = 0.0 \quad \mu_{\text{middle-aged}}(53) = 0.8 \quad \mu_{\text{old}}(53) = 0.2 \]

Region No. = 2 Region No. = 2 Region No. = 0

Since the value 53 belongs to the fuzzy term middle-aged with the maximum MD among all fuzzy terms, the fuzzy term middle-aged is used while constructing the BS. We then construct the BS as shown in Table II, according to the scheme explained in Section III-B.

Finally, the start and stop BSs for the crisp exact match query become

Start BS: 010100111011010101
Stop BS: 010100111011010101.

**Crisp Range Query:** Crisp range conditions are in the form

\[ \text{indexed attribute} \geq \text{crisp attribute value} \]

We calculate the fuzzy term to which the crisp value mostly belongs, its region number and its MD. Then, for each single valued fuzzy term, the start BSs and stop BSs are constructed using these parameters according to the range condition.
Begin
Do forward traversal on the path P starting from object X
Obtain multidimensional values contained in the nested
organizing attributes of the object X
Set $S =$ the set of multidimensional values obtained
for each multidimensional value $Val \in S$
do
Construct a bit string for $Val$
Set $BS Val = $ the bit string constructed
end for
for each bit string $BS \in BSs$
do
Access the data bucket $DB$ using the $BS$
if $DB$ is null then
Continue // nothing to delete
end if
Access the data bucket record $DBR$ whose bit string is $BS$
if $DBR$ is null then
Continue // nothing to delete
end if
Access the Path Instantiation Node $PIN$ of $DBR$
Update the path instantiations in $PIN$ so that they do not
contain object $X$ anymore
Do redistribution and merge operations on leaf and
non-leaf nodes if required
end for
end

Fig. 17. Deletion algorithm of the FI structure.

Begin
Determine the type of the query
Construct the start and stop bit string pairs
according to the query type
Set $BSL = $ the list of the start and stop bit
string pairs constructed
for each bit string pair $BSP \in BSL$
do
Set $PIS = \{\}$ // the set of path instantiations
Access the data bucket $DB$ using the start
bit string of $BSP$
while the bit string of $DB$ is between start
bit string and stop bit string of $BSP$
do
Access the data bucket record $DBR \in DB$
Access the data bucket record
if the bit string of $DBR$ is between start
bit string and stop bit string of $BSP$
Access the path instantiation node $PIN$ of $DBR$
for each path instantiation $PI$ in $PIN$
do
if $PI$ satisfies from condition of the query then
$PIS = PIS \cup PI$
end if
end if
end if
end for
$DB = $ the following data bucket of $DB$
end while
end for
Return the objects belonging to the target class in the query
within the path instantiations in $PIS$
end

Fig. 18. Retrieval algorithm of the FI structure.

For example, if we assume that the *where* condition is “age $< 32$” and if we use the membership functions given in Fig. 19, then we calculate the MDs and region numbers of the attribute value 32 for the fuzzy terms as follows:

- $\mu_{young}(32) = 0.2$
- $\mu_{middle-aged}(32) = 0.8$
- $\mu_{old}(32) = 0.0$

Since the condition is “age $< 32$” and 32 has the highest MD (0.8) for the fuzzy term *middle-aged*, we have to deal with the values *middle-aged* and *young*. Considering the regions, we have to search the search spaces given in Table III.

From these search spaces, we construct the BS pairs given in Table IV, as was done for the example given earlier for the crisp exact match query.

**Fuzzy Query:** Fuzzy conditions are conditions in the form

\[ \text{indexed_attribute} = \text{fuzzy\_term} \cdot \text{threshold\_value}. \]

The fuzzy condition contains an equality followed by a fuzzy term and a threshold value. For example, “age = [middle-aged] 0.8” and “height = [short, middle] 0.5” are fuzzy conditions in the case where those attributes *age* and *height* are the organizing attributes. The threshold value provides flexibility for users when specifying the degree of uncertainty and eliminates unnecessary data from the results of queries.

We calculate the degrees of similarity among the fuzzy terms in the domain. Then, for each single or multivalued fuzzy term, the start BSs and stop BSs are constructed using those parameters according to the range condition.
For example, if we assume that the where condition is “Age = [middle-aged] 0.6”, and if we use the membership functions to construct BS pairs and the fuzzy term in the condition is a single valued fuzzy term, then we use the similarity matrix given in Table V and the multiplication operator:

\[ \mu_{\text{middle-aged}}(x) = 0.6 \Rightarrow \mu_{\text{young}}(x) = 0.6 \times 0.6 = 0.36 \]

\[ \mu_{\text{middle-aged}}(x) = 0.6 \Rightarrow \mu_{\text{old}}(x) = 0.6 \times 0.5 = 0.3. \]

In Fig. 19, it can be seen that any crisp value that is a member of the fuzzy term young and is in Region2 with an MD between 0 and 0.36 (similar to middle-aged) should be retrieved for the given condition. In a similar way, we should also retrieve the objects with the value old in Region0 with an MD between 0 and 0.3.

In the fuzzy age domain, we have three fuzzy terms: young, middle-aged, and old. We can have seven different single and multivalues from these three terms: \{young\}, \{middle-aged\}, \{old\}, \{young, middle-aged\}, \{young, old\}, \{middle-aged, old\}, \{young, middle-aged, old\}. Using the similarity matrix given in Table V and the division operator, we calculate the following threshold values for any single or multifuzzy-valued attributes. Remember that the condition was “age = [middle-aged] 0.6”

\[ \mu_{\text{middle-aged}}(x) = 0.6 \Rightarrow \mu_{\text{young}}(x) = \frac{0.6}{0.6} = 1.0 \]

where the numerator 0.6 is the threshold given in the query condition and the denominator 0.6 is the similarity of fuzzy terms young and middle-aged and taken from Table V. The thresholds of other single values are calculated in a similar manner:

\[ \mu_{\text{middle-aged}}(x) = 0.6 \Rightarrow \mu_{\text{middle-aged}}(x) = \frac{0.6}{1.0} = 0.6 \]

\[ \mu_{\text{middle-aged}}(x) = 0.6 \Rightarrow \mu_{\text{old}}(x) = \frac{0.6}{0.5} = 1.2. \]

For multivalues:

\[ \mu_{\text{middle-aged}}(x) = 0.6 \Rightarrow \]

\[ \mu_{\text{middle-aged}}(x) = 0.6 \Rightarrow \mu_{\text{middle-aged}}(x) = \frac{0.6}{0.6} = 1.0. \]

where the numerator is the threshold given in the query condition and the denominator is the minimum of the similarities of each fuzzy term to the fuzzy term in the query condition. That is:

\[ \min(S(\text{young}, \text{middle-aged}), S(\text{middle-aged}, \text{middle-aged})) = \min(0.6, 1) = 0.6. \]

Similarly,

\[ \mu_{\text{middle-aged}}(x) = 0.6 \Rightarrow \mu_{\text{young,old}}(x) = \frac{0.6}{0.5} = 1.2 \]

\[ \mu_{\text{middle-aged}}(x) = 0.6 \Rightarrow \mu_{\text{middle-aged,old}}(x) = \frac{0.6}{0.5} = 1.2. \]

If the threshold value is specified as 0.6 in the condition, we have to consider the values whose membership values are between 0.6 and 1.0. Therefore, we need to seek five search spaces given in Table VI; two for crisp values and three for fuzzy values. From these search space intervals, we construct the BS pairs given in Table VII.

**IV. PERFORMANCE EVALUATIONS**

In this section, we briefly present the performance test results. Since the FI is the first and the only indexing technique proposed so far for the FOOD model, the FI is compared with the path index structure, which is currently accepted as one of the most efficient index structure (to our best knowledge) for the object-oriented database model [4]. We tested the FI and the path index with special implementations. The performance evaluations were done by considering the performances of insertion and deletion operations and retrieval performances of various types of queries. We have repeated the performance tests for four different database sizes (25,000, 50,000, 75,000, and 100,000 objects) using synthetic data. Our approach for performance evaluations is similar to the guidelines included in [9] and [10]. Since the proposed index structure supports both fuzzy and crisp data, we produced approximately 50% fuzzy and 50% crisp data for fuzzy attributes.

**A. Insertion and Deletion**

We have carried out a number of performance tests for the insertion of 25,000, 50,000, 75,000, and 100,000 synthetic objects into the database. When an object was to be inserted into the
database, first a random number was generated to determine the class of the object. Next, other random values were generated for the object values, and then, the object was created. After insertion of the different numbers of objects for the FI and the PI structures, the results given in Fig. 20(a) were obtained.

For the deletion operation, starting with four FOOD databases that contained 25,000, 50,000, 75,000, and 100,000 objects, all of the objects were deleted from databases randomly. We compared the performances of the FI and the path index structures as shown in Fig. 20(b).

Fig. 20(a) and (b) shows that the FI has a better performance than the path index for insertion and deletion operations. The reason is the reorganization of the PIN in the FI, as explained in Section III-A.

B. Retrieval

When testing the performance of retrieval operations, we considered three types of retrieval queries: crisp exact match query, crisp range query, and fuzzy query. Initially, we had four FOODs containing 25,000, 50,000, 75,000, and 100,000 objects. We ran a query for each type of retrieval query to compare the performances of the retrieval operation for the FI and the PI index. In addition, performance tests for the retrieval operations in 1-D (indexed on single attribute), 2-D (indexed on two attributes), and 3-D (indexed on three attributes) cases in the FI were carried out and compared with each other.

**Crisp Exact Match Queries:** The results of the performance tests for the crisp exact match query are shown in Fig. 21(a). The figure shows that the FI has a slightly better performance than the path index for crisp exact match queries. Again, the reason is the reorganization of the PIN adapted for the FI. The crisp exact match queries for the FI in different dimensions are compared in Fig. 21(b). Since crisp exact match queries refer to a point in space, the numbers of I/O for each dimension are similar to each other. The difference between them arises from the depth of the tree and the fullness of the data buckets. Also, the number of I/Os is affected by the number of the PIs in the PIN accessed. When the dimension increases, the number of different key values also increases. It causes an increase in the fullness of the data buckets, but a decrease in the average number of the PIs for each PIN. Therefore, depending on the average number of PIs for each PIN and the depth of the tree, slight differences (an increase or a decrease) may occur when the dimension increases.

**Crisp Range Queries:** The selectivity of the range queries that we have used is approximately 50%. The results of the performance tests are shown in Fig. 22(a). Once again, the figure shows that the FI has a better performance than the path index for crisp range queries. The reason for this improvement is once more the reorganization of the PIN adapted for the FI. The crisp range queries for the FI in different dimensions are compared in Fig. 22(b). When the dimension increases, the number of I/O increases. This is because in addition to the increase in the number of different key values, the search space for the range query increases with new dimension(s), which causes the performance of this kind of query to decrease, as expected.
values together with crisp attribute values. The path index does a linear search between fuzzy values to see whether or not they satisfy the query conditions, and this costs much more than the FI requires.

The fuzzy queries for the FI in different dimensions are compared in Fig. 23(b). Similar to the crisp range queries, the search space for the fuzzy query increases with the new dimension(s) while the performance decreases.

V. CONCLUSION

In this study, we introduced a multidimensional indexing technique (the FI structure) to efficiently handle both fuzzy and crisp queries. It can be used for both aggregation and inheritance hierarchies and to deal with the fuzzy relations of the FOOD model. After constructing the FI structure, we investigated the retrieval, insertion, and deletion algorithms.

Having completed all the performance tests, we may conclude that the FI has not only a considerably better performance than the path index for fuzzy queries, but also a slightly better performance than the path index for insertion and deletion operations, crisp exact match and crisp range queries. One reason is that the FI is specially designed to handle fuzzy queries efficiently, and the other reason is that the PIN of the path index is reorganized. In the FI, PINs are organized according to their lengths, and when there is a query on a path, the length of the path can be determined easily, and this information is used to access related objects more efficiently. This improvement results in a slightly better performance of the FI, compared to that of the path index, for crisp exact match and crisp range queries.

Further research is required to adapt the FI structure to spatiotemporal and multimedia database applications. Research aimed at developing a FOOD management system and use of the FI in such a system for some real life applications needs to be carried out in the future.

REFERENCES


Adnan Yazici (M’89–SM’00) received the Ph.D. degree in computer science from the Department of Electrical Engineering and Computer Science, Tulane University, New Orleans, LA, in 1991.

During 1998–2000, he was a Visiting Professor at Tulane University. He is currently with the Department of Computer Engineering, Middle East Technical University, Ankara, Turkey. He is the author or coauthor of more than 100 published international technical papers and is the coauthor of Fuzzy Database Modeling (Springer-Verlag, 1999). He has also edited a Lecture Notes in Computer Science (LNCS) on computer and information sciences. He serves as a committee member and reviewer in various international journals and conferences. His current research interests include intelligent database systems, fuzzy database modeling, design and analysis of algorithms, indexing, spatiotemporal databases, multimedia and video databases, and wireless sensor networks.

Dr. Yazici is the recipient of the Young Investigator Awards bestowed by the Parlar Foundation, for the year 2001. He co-chaired the 23rd IEEE International Conference on Data and Knowledge Engineering (ICDE) in 2007, and was the Conference and Program Committee Chair of the 18th International Symposium of Computer and Information Sciences (ISCIS 2003).

Cagri Ince received the B.S. degree from the Eastern Mediterranean University, Gazimagusa, North Cyprus, and the M.S. degree from the Middle East Technical University, Ankara, Turkey, in 1999 and 2003, respectively, both in computer engineering.

He was a Software Engineer with several technology companies such as Havelsan and Cybersoft in Ankara, Turkey. He is currently with the Department of Computer Engineering, Middle East Technical University, Ankara. His current research interests include indexing in database systems and different aspects of fuzziness in database systems.

Murat Koyuncu received the B.S. degree in electronics, the M.S. degree in computer engineering, and the Ph.D. degree in computer engineering from the Middle East Technical University, Ankara, Turkey, in 1986, 1995, and 2001, respectively.

He is currently an Assistant Professor in the Department of Computer Engineering, Attilim University, Ankara. His research interests include fuzzy logic, object-oriented databases, knowledge-based systems and multimedia databases.