Designing nesting structures of user-defined types in object-relational databases

Wai Yin Mok *

Department of Economics and Information Systems, University of Alabama in Huntsville, Huntsville, AL 35899, USA

Received 1 June 2005; received in revised form 14 October 2006; accepted 24 October 2006
Available online 29 November 2006

Abstract

This paper presents a methodology for designing proper nesting structures of user-defined types in object-relational databases. Briefly, we envision that users model a real-world application by using the EER model, which results in an EER schema. Our algorithm then uses the theory we developed for nested relations to generate scheme trees from the EER schema. We shall prove that the resulting scheme trees have exactly the same information content as the EER schema, and the scheme-tree instances over the resulting scheme trees do not store information redundantly. Finally, the scheme trees are transformed to Oracle Database 10g nested object types for implementation. The algorithm in this paper forms the core of a computerized object-relational database design tool we shall develop in the future.

Keywords: The EER model; Object-relational databases; Oracle database 10g nested object types; Scheme trees

1. Introduction

Object-relational databases are becoming commonplace because many commercial database management systems are adding object-oriented capabilities to their products. Some examples are Oracle Database 10g, IBM DB2, and PostgreSQL. A common feature of these new database management systems is that they all support SQL: 2003 [4] to a certain degree. Under this new standard of SQL, user-defined types (UDTs), which are a major feature of SQL: 2003, have been added to SQL. The SQL: 2003 object model allows the type of a field in a UDT to be another UDT. As a result, SQL: 2003 allows nested UDTs. In other words, UDTs can be defined within another UDT.

This paper presents a methodology that designs proper UDT nesting structures in object-relational databases. Our goal is to automate our object-relational database design methodology by developing a computerized design tool. Our methodology envisions that users model a real-world application with the Extended ER (EER) model [1], which results in an EER schema of the real-world application of interest. From the EER schema, the scheme-tree generation algorithm in this paper generates a collection of scheme trees [13,15], which are simple hierarchical structures. We shall prove in this paper that our scheme-tree generation algorithm guarantees that the resulting scheme trees have exactly the same information content as the EER schema, and the scheme-tree instances over the resulting scheme trees do not store information redundantly. Finally, we present a straightforward mapping from scheme trees to Oracle Database 10g nested object types for actual implementation of the resulting scheme trees. The scheme-tree generation algorithm and the mapping from scheme trees to Oracle Database 10g nested object types in this paper form the core of the design tool we shall develop in the near future.

A large body of theory has been developed in the literature to translate a conceptual data model to an implementation data model. For example, using UML [2], Refs. [11,12] define a methodological approach for object-relational database design. Specifically, the authors present a
methodology that defines new UML stereotypes for object-relational databases and propose several guidelines to translate an UML schema into an object-relational one. The guidelines are based on the SQL:1999\(^2\) object-relational model and they use Oracle8i as a target DBMS. In their papers, they focus on the UML extensions required for object-relational database design.

Ref. [5] introduces a methodology in which a relational schema is translated into an Extensible Markup Language (XML) schema of an XML database that is simple and efficient for the Internet. The authors first reverse engineer a relational schema into an EER schema. Afterward, using semantic transformation the EER schema is mapped to an XML Schema Definition Language (XSD) graph, which is an XML conceptual schema. Finally, the XSD graph is forward engineered into an XML logical schema written in XSD, and at the same time the semantics of participation, cardinality, generalization, aggregation, categorization, \(N\)-ary and \(U\)-ary relationships are all preserved in the process.

The focus of [6] is the transformation of conceptual data models (such as ER, NIAM and PSM) to object-oriented databases. Conceptual models are first mapped to abstract intermediate specifications, which are then transformed to database schemas in a given object-oriented database environment. In this way, the authors can treat different target systems uniformly. As the final implementation environments, the authors consider object-oriented as well as object-relational DBMSs, including the SQL:1999 (also called SQL3) and ODMG-93 standards.

In [3], the authors investigate a practical framework for accessing heterogeneous data sources. The framework utilizes XML schema as the canonical data model for querying and integration of data from heterogeneous data sources. They provide algorithms for mapping relational and network schemas into XML schemas. Finally, a system prototype for heterogeneous database access is also presented.

Ref. [17] presents a transformation methodology from inheritance relationships to relational tables. This includes transformation of different types of inheritance, such as union inheritance, mutual exclusion inheritance, partition inheritance and multiple inheritance. The authors also compare their proposed transformation methodology and existing methods. Based on their preliminary evaluation, they conclude that their proposed transformation methodology is more efficient than the others. In [18], the authors present a more rigorous evaluation. This time it covers I/O cost models of different types of queries. The evaluation is basically comparison-based, in which the performance of SQL operations upon a set of tables derived from the relational data model is compared with the tables derived from the object-oriented data model using their transformation methodology. Ref. [18] further substantiates that the performance of the relational database implementation transferred from an object-oriented model using their object-relational transformation methodology is better than the relational implementation using a conventional relational modeling.

The authors of [20] present an approach to the symbiosis of the object-oriented and relational data models, which is built into GiniSNT – an object-oriented platform for the development of scalable, end-user GIS applications. The mapping algorithm transforms classes and objects into relations and tuples, and vice versa, instantiates objects from relational databases. The methodology presented in [20] is claimed to be extremely efficient, as has been substantiated by a number of applications developed in GiniSNT, and is at the same time cost efficient.

This paper is different from these papers because our methodology makes use of the theory we developed for nested relations [13,15] to produce proper UDT nesting structures in object-relational databases. Particularly, to the best of our knowledge, no research that we are aware of uses NNF (Nested Normal Form) as a guide for designing proper nesting structures. The central idea of NNF is that it is a precise characterization of redundancy-free nested relations with respect to a given set of multivalued dependencies (MVDs) and functional dependencies (FDs) [10]. We have published a preliminary paper on making use of NNF to design object-relational databases in the past. In [16], we presented a sketch of an algorithm that transforms a UML class diagram to a set of SQL:1999 create-table statements. That algorithm first generates a semantically equivalent UML class diagram that is acyclic from the input UML class diagram. Then, it uses the algorithm in [13] to generate a set of NNF scheme trees from the acyclic UML diagram, and eventually to a set of SQL:1999 create-table statements. The algorithm in this paper, however, does not have to first generate an acyclic EER schema from the input EER schema. In this sense, the algorithm of this paper simplifies the one in [16]; and in addition, it is rigorously defined, as opposed to the sketch in [16].

A precondition of our methodology is that the given EER schema is redundancy-free. Normalization in ER schemas has been studied extensively. For example, Refs. [8,9,19] present algorithms that remove redundancy in an ER schema and generate a semantically equivalent ER schema for further processing. Specifically, Ref. [8] defines a normal form for ER schemas and transformation rules for such a normal form. When the resulting ER schema is mapped to a relational database, 3NF and 5NF relation schemes are obtained [10]. Ref. [9] extends the work in [8] to object-oriented ER schemas. Finally, the authors of [19] further extends this idea to define a normal form for ER schemas, called ER-BCNF; and they define transformation rules to obtain an ER schema in such a normal form. Based on these researches, we are safe to assume that the input ER schema to the algorithm in this paper has been processed so that it has no redundancy. Thus, the goal

\(^2\) SQL:1999 is a previous version of SQL: 2003.
of our algorithm is to create scheme trees without redundancy from a given redundancy-free ER schema.

This paper is organized as follows. Section 2 presents the concepts of the EER model that are relevant to this paper, the formal definitions of scheme trees and scheme-tree instances, and what we mean by information content preservation — a condition that must be met as we translate an EER schema to scheme trees. Section 3 sets forth four requirements that can lead to information content preservation and presents the algorithm of this paper. We shall prove in Section 3 that the resulting scheme trees of our algorithm preserves information content of the input EER schema and the scheme-tree instances do not have redundancy. Section 4 shows how scheme trees can be mapped to Oracle Database 10g nested object types for actual implementation. We conclude and point out future work in Section 5.

2. Basic definitions

In this section, we introduce the concepts provided by and the respective graphic representations used in the EER model. We basically follow [1]. There are many variations of the EER model [7]; each has its own strengths and weaknesses. The reason we follow [1] is because some of the concepts in [1], such as minimal and maximal cardinalities and generalization/specialization, are included in UML as well. Here, we also present the definitions of scheme trees and scheme-tree instances.

2.1. The EER model

Because most of the concepts of the EER model are well-understood, we introduce the concepts that are relevant to this paper by examples rather than by formal definitions.

Entity types. An entity type, which is a class of real-world objects, is graphically represented as a rectangle, like male and female in Fig. 1. We are interested in two kinds of entity types in this paper: strong entity types and weak entity types. A strong entity type has an internal identifier (described below). A weak entity type, on the other hand, derives its identifier from the strong entity types on which it depends.

Attributes. An attribute of an entity type is represented as a small circle with a line connected to the rectangle for the entity type, as shown in Fig. 1. Each attribute has a pair of associated numbers (min-card, max-card), where min-card means the minimal cardinality and max-card means the maximal cardinality of the attribute. For example, a female person can have 0 or \( n \) \( (n \geq 0) \) hobbies; whereas a male person can have 0 or 1 draft status. The (min-card, max-card) pair (1,1) arises so frequently that (1,1) is the default for an attribute. For example, since the (min-card, max-card) pair is omitted for the attribute name of the entity type person, it is assumed to be (1,1), which means a person must have a single name and it cannot be a null. Since (1,1) is the default, the pair (1,1) for SSN is in fact redundant. Note that the circle for SSN is solid, which means it is an identifier for the entity type person. Attributes can be categorized as single-valued, multivalued, optional, mandatory and composite. We can determine whether an attribute is single-valued or multivalued by examining its max-card number. For example, since the max-card number of the attribute draft_status is 1, draft_status is a single-valued attribute of the entity type male. Since the max-card number of the attribute hobby is \( n \), hobby is a multivalued attribute of the entity type female. We can also determine whether an attribute is optional or mandatory by examining its min-card number. The min-card number of an optional attribute is equal to 0 while the min-card number of a mandatory attribute is not equal to 0. For example, draft_status is an optional attribute for the entity type male, which means a male person does not have to have a draft status. On the other hand, SSN is a mandatory attribute for the entity type person, which means a person must have a SSN. Address in the EER schema in Fig. 1 is a composite attribute of the entity type person, which consists of three (non-composite) attributes street, city and state.

Relationships. Relationships represent aggregations of two or more entity types. Dated and married_to are two different relationships between the entity types male and female in the EER schema in Fig. 1. A relationship
also has a \((\text{min-card, max-card})\) pair for each participating entity type. For the relationship \(\text{dated}\), it is \((0, n)\) for \text{male} and \((0, n)\) for \text{female}. This means that a male person can participate 0 or more times in the relationship \(\text{dated}\). In other words, a male person is allowed to have \(\text{dated}\) 0 or more female persons. The same is true for a female person. For the relationship \(\text{married_to}\), however, the \((\text{min-card, max-card})\) pair for both \text{male} and \text{female} is \((0, 1)\), which means a male person is at most married to a single female person and a female person is at most married to a single male person. Like attributes, an entity type's participation in a relationship can be categorized as \text{optional} and \text{mandatory}. If the \text{min-card} number of an entity type in a relationship is 0, the entity type participates optionally in the relationship. This means an instance of the entity type does not have to participate in the relationship. For example, because the \text{min-card} number of \text{female} in \(\text{married_to}\) is 0, there might be a female person who is unmarried to any male person. On the other hand, if the \text{min-card} number of an entity type in a relationship is not equal to 0, the entity type participates mandatory in the relationship. This means every instance of the entity type must participate in the relationship.

\textbf{Generalization hierarchies.} A \text{generalization} defines a subset relationship between two entity types. There is a simple generalization hierarchy in Fig. 1 in which \text{person} is the generalization and \text{male} and \text{female} are the specializations. A fundamental property of a generalization hierarchy is that the specializations inherit all the abstractions defined for the generalization, which means both \text{male} and \text{female} inherit the attributes \text{SSN}, \text{name}, and \text{address} from \text{person} in the EER schema in Fig. 1. Since \text{SSN} is an identifier for \text{person}, it is also an identifier for \text{male} and \text{female} as well.

2.2. \text{Scheme} trees

The scheme-tree definition in this paper is different from the one in [15]. In [15], we presented a normal form, called NNF, which has the property that if a scheme tree is in NNF, then all scheme-tree instances over the scheme tree do not have data redundancy with respect to the given FDs and MVDs that hold for the scheme tree. In [13], we presented an algorithm that generates NNF scheme trees from an acyclic hypergraph [10] and a set of FDs where each given FD is contained in an hyperedge and each hyperedge is in BCNF. Despite these earlier publications, we need a different definition of scheme trees for this paper because [13,15] are based on the relational database theory [10]. This theory, however, requires that each attribute (column) of a relation (table) is unique. In other words, no attribute can appear more than once in a relation. This follows that for any set of attributes, there is a unique relationship among the attributes in that set. Such a requirement is too restricted for the ER model. In fact, it is natural to have two different relationships between two different entity sets in an ER schema. For example, male persons and female persons are naturally related in more than one way. Thus, in the EER schema in Fig. 1, we can see that \(\text{dated}\) and \(\text{married_to}\) are two different relationships between \text{male} and \text{female}. To map these different relationships to scheme trees, one possible way is to satisfy the requirement of the relational database theory by adding more attributes so that each attribute is unique in a scheme tree. Another way, however, is not to satisfy this requirement and thus allowing an attribute to appear more than once in a scheme tree. This way allows a more straightforward mapping between ER schemas and scheme trees, which is the approach adopted in this paper. Consequently, denoting different semantic relationships among the attributes in a scheme tree becomes an implementation issue. In this paper, we use data types for this purpose which is exemplified in Section 4. For example, in Example 9, different data types are used to denote the different relationships between the attributes \text{title} and \text{SSN}.

Even though the scheme-tree definition in this paper is different from that of [13,15], the idea of NNF is still applicable to our design methodology. Particularly, the redundancy-free idea of NNF leads us to require that every instance of each entity type and relationship must only be represented once in a scheme-tree instance (which will be defined shortly); otherwise, we would have the problem of redundant storage [14]. This point would become clear after we present several examples in Section 2.3. We now give definitions for scheme trees and scheme-tree instances that are tailored especially for this paper.

\textbf{Definition 1.} Let \(U\) be a set of attributes. A \text{scheme tree} \(T\) over \(U\) is a rooted tree where every node in \(T\) is a nonempty subset of \(U\). Let \(T\) be a scheme tree and \(N\) be \(T\)'s root node. Further let \(N_1, N_2, \ldots, N_n\) \((n \geq 0)\) be the child nodes of \(N\) and \(T_1, T_2, \ldots, T_n\) be the subscheme trees of \(T\) that are rooted at \(N_1, N_2, \ldots, N_n\), respectively. We can also write \(T\) as \(N_r (T_1)^* (T_2)^* \ldots (T_n)^*\) as well.

The main difference between Definition 1 and the scheme-tree definition in [13,15] is that the nodes of a scheme tree that satisfies Definition 1 are not necessarily pairwise disjoint. In other words, an attribute might appear in more than one tree node. However, this is not the case for the scheme trees in [13,15].

\textbf{Definition 2.} Let \(T\) be a scheme tree over a set \(U\) of attributes. A \text{scheme-tree instance} \(r\) over \(T\) is a set of nested tuples recursively defined as follows:

1. If \(T\) has only the root node \(N_r\), then \(r\) is a flat relation over \(N_r\). Each row in \(r\) is called a \text{nested tuple} over \(T\).
2. If \(T\) has more than one node, let \(N_r\) be the root node of \(T\) and \(N_1, N_2, \ldots, N_n\) \((n \geq 1)\) be the child nodes of \(N_r\). Further let \(T_1, T_2, \ldots, T_n\) be the subscheme trees of \(T\) that are rooted at \(N_1, N_2, \ldots, N_n\), respectively. \(r\) is a \text{nested tuple} in \(r\) if the following conditions hold:
For each attribute \( A \in N_r \), \( t(A) \) is in the domain of \( A \) or \( t(A) \) is a null and \( t(T_i) \) is a scheme-tree instance over \( T_i \) (\( 1 \leq i \leq n \)). Further, if \( t(A) \) is a null for every \( A \in N_r \), \( t(T_i) \) must be empty (\( 1 \leq i \leq n \)).

(a)

(b) If \( t' \) is another nested tuple in \( r \) such that \( t(N_r) = t'(N_r) \), then \( t(T_i) = t'(T_i) \) (\( 1 \leq i \leq n \)). Consequently, \( t = t' \).

The main difference between Definition 2 and the scheme-tree instance definition in \([13,15]\) is that scheme-tree instances in this paper allow nulls while those of \([13,15]\) do not.

### 2.3. Information content preservation

In this section, we illustrate what we mean by information content preserving transformation from an EER schema to a set of scheme trees. In such a transformation, a minimum requirement is that every instance of each entity type and relationship must be represented by some nested tuples. However, if an instance of an entity type or a relationship is represented by only one nested tuple, then we would have redundant storage. Examples are helpful to provide an overview on the concept of information content preservation.

**Example 1.** This example presents a legal instance over the EER schema in Fig. 1, which will be referenced later in this paper. We use the notation of \([1]\) for this example. Suppose \( EER \) schema in Fig. 1, which will be referenced later in this example presents a legal instance over the EER schema.

To preserve the instances of a relationship, every participating entity type of the relationship must have an identifier before we generate scheme trees from an EER schema. **Example 2** demonstrates what would happen if this condition was not satisfied.

**Example 2.** Fig. 2(a) shows a scheme tree and Fig. 2(b) shows an illegal scheme-tree instance over the scheme tree in Fig. 2(a). The scheme-tree instance in Fig. 2(b) is illegal because there are two nested tuples \( t_1 \) and \( t_2 \) such that \( t_1(\text{married}) = t_2(\text{married}) = \text{drafted} \) but \( t_1(\text{maiden_name}(\text{hobby}^*)) \neq t_2(\text{maiden_name}(\text{hobby}^*)) \). This results in a violation of Definition 2. However, the problem of the scheme-tree instance in Fig. 2(b) is more than this violation. The intention of the scheme tree in Fig. 2(a) is to show how the attribute values of a male person relate to the attribute values of the female person to whom he is married. According to the legal instance in **Example 1**, male persons \( m_2 \) and \( m_3 \), respectively, are married to female persons \( f_2 \) and \( f_3 \) and their draft status, maiden names and hobbies are indeed shown in Fig. 2(b). However, since the identifiers of \( m_2, m_3, f_2 \) and \( f_3 \) are missing in Fig. 2(b), we cannot determine who is married to whom. In other words, the scheme tree in Fig. 2(a) does not preserve the relationship \( \text{married_to} \).

The scheme tree in Fig. 3(a) attempts to correct the problem of the scheme tree in Fig. 2(a) by incorporating the identifiers for male and female. However, it introduces the problem of redundant storage, as shown below.

**Example 3.** To correct the problem of the scheme tree in Fig. 2(a), we explicitly add the identifiers for the entity types male and female, which are inherited from person, before we generate scheme trees. The scheme tree in Fig. 3(a) is a possible attempt. There are two identical subscheme trees in Fig. 3(a) and both of them are rooted at \( \text{SSN} \), \( \text{Maiden_name} \). For a particular male person, the nested tuples over the left subscheme tree are meant to store the values of \( \text{SSN}, \text{maiden_name} \), and hobby of the female persons whom he dated whereas the nested tuples over the right subscheme tree are meant to store the values of \( \text{SSN}, \text{maiden_name} \), and hobby of the female person to whom he is married. As we can see in the scheme-tree instance in Fig. 3(b), the relationships dated and \( \text{married_to} \) are preserved. However, because there

![Fig. 2. A scheme tree that does not preserve the relationship married_to.](image-url)
are two different male persons who have dated Mary Shaw (whose SSN is 555-66-7777), her SSN, maiden name and hobbies are stored redundantly in two different nested tuples over the left subscheme tree. Even worse, since Mary Shaw is married to Dan Shaw, the same set of values also appears in a nested tuple over the right subscheme tree. Because of the same reason, the attribute values of Sue Carson (whose SSN is 666-77-8888) also appear redundantly.

The scheme tree in Fig. 4(a) attempts to correct the problem of the scheme tree in Fig. 3(a). Although it is a better design, it does not preserve the entity type female, as shown below.

**Example 4.** The scheme tree in Fig. 4(a) attempts to avoid the problem of the scheme tree in Fig. 3(a) by removing the attributes maiden name and hobby from the left subscheme tree. Now the nested tuples over the left subscheme tree only store the social security numbers of the female persons whom a male person dated. However, such a scheme tree introduces another problem. Because Ann Gale (whose SSN is 444-55-6666) is not married to any male person, her hobby does not appear anywhere in the scheme-tree instance in Fig. 4(b). Although the scheme tree in Fig. 4(a) preserves both relationships dated and married to and the entity type male, it does not preserve the information content of the entity type female.

A correct solution to preserve the relationships dated and married to and the entity types male and female and to avoid the redundancy problem of Example 3 requires two scheme trees, as shown below.

**Example 5.** The scheme-tree forest in Fig. 5(a) is a correct design to preserve both relationships dated and married to and both entity types male and female. By using two scheme trees instead of one, we avoid the redundancy problem of Example 3. Now all male persons and female persons are represented by nested tuples over the two scheme trees and all the relationship instances of dated and married to are also accounted for.

In fact, the scheme-tree forest in Fig. 5(a) has an additional advantage. Consider the case that there was a female person who had never dated a male person nor been married to anybody. In such a case, her SSN would not appear in the first scheme-tree instance in Fig. 5(b). However, her SSN and hobbies would appear in the second scheme-tree instance in Fig. 5(b). Thus, there would be no loss of information content. The same argument also applies to a male person who had never dated a female person nor been married to anybody.
3. Scheme tree generation

Recall that in Section 1 we emphasize that the input EER schema to our scheme-tree generation algorithm is assumed to have been processed earlier in order to remove all redundant information. Techniques for removing redundant information in an EER schema have been published in the literature. Thus, the main purpose of our scheme-tree generation algorithm is to generate redundancy-free and information content preserving scheme trees from an EER schema that has no redundant information.

Based on our previous work [14], given an EER schema as the input, our scheme-tree generation algorithm must satisfy the following requirements to ensure that the resulting scheme trees of our algorithm preserve every relationship and every entity type in the input EER schema and will not lead to redundant storage.

**Requirement 1.** For every instance $e$ of an entity type $E$, there is a nested tuple $t$ in a scheme-tree instance such that $t$ stores the information of $e$.

**Requirement 2.** For every instance $r$ of a relationship $R$, there is a nested tuple $t$ in a scheme-tree instance such that $t$ stores the information of $r$.

**Requirement 3.** Every subset relationship is enforced by a referential integrity constraint.

**Requirement 4.** If $t_1$ and $t_2$ are nested tuples such that (1) $t_1$ is not a subnested tuple of $t_2$, (2) $t_2$ is not a subnested tuple of $t_1$, and (3) both $t_1$ and $t_2$ store the information of an instance of an entity type or a relationship, then it must be the case that $t_1$ and $t_2$ are the same nested tuple.

Requirements 1 and 2 stipulate that every instance of an entity type or a relationship must be stored in a scheme-tree instance. Requirement 3 states that the subset relationships of a generalization hierarchy must be enforced. Requirement 4, however, states that for each instance of an entity type or a relationship, there is a unique nested tuple that stores the information of such an instance. To satisfy all of these requirements, we first introduce a special type of scheme trees, called ID scheme trees.

**Definition 3.** An ID scheme tree $T$ is a scheme tree where every node in $T$ is an identifier of an entity type.

**Example 6.** None of the scheme trees we have presented so far is an ID scheme tree. Nevertheless, if we remove the attribute $\text{draft_status}$ from the scheme tree $SSN \, \text{Draft_status} \, (SSN)^*$ in Fig. 5(a), we will obtain an ID scheme tree where the root node contains the attribute $SSN$ of $\text{male}$ and the other two nodes contain the attribute $SSN$ of $\text{female}$, as shown in Fig. 6.

The purpose of an ID scheme tree is to capture relationships. The ID scheme tree in Fig. 6 captures the two relationships $\text{dating}$ and $\text{married_to}$ in the EER schema in Fig. 1. A scheme-tree instance over the ID scheme-tree in Fig. 6, which is constructed from the legal instance in Example 1, is shown in Fig. 7. Note that every instance of the two relationships $\text{dating}$ and $\text{married_to}$ is accounted for.

### 3.1. The definition of the algorithm

Note that since there are so many variations of the EER model, it is almost impossible to define an algorithm that considers all different possibilities. Thus, we limit our discussion to the scope of the EER model defined in Section 2.1. Since the central idea of our scheme-tree
generation algorithm is to ensure that each instance of an entity type or a relationship is stored in a unique nested tuple, this idea can be used to expand our scheme-tree generation algorithm to any new variation of the EER model.

Our scheme-tree generation algorithm is divided into two stages, which are implemented, respectively, by Procedure 1 and Procedure 2. Procedure 1 constructs ID scheme trees from the relationships in the input EER schema. Procedure 2 adds the attributes of the entity types and relationships in the input EER schema to the resulting ID scheme trees. We first cover Procedure 1. In constructing ID scheme trees, we emphasize that Procedure 1 does not use any FDs that are not implied by the (min-card, max-card) pairs of participating entity types in relationships of the input EER schema. In other words, Procedure 1 only makes use of the (min-card, max-card) pairs of participating entity types in relationships.

**Procedure 1.**

Input: a redundancy-free EER schema $M$
Output: an ID scheme-tree forest

Step 1 Prepare the input EER schema $M$:
1.1 Unmark every relationship and every entity type in $M$. In addition, reduce each composite attribute in $M$ to its component attributes.
1.2 For each specialization in a generalization hierarchy in $M$, make the specialization inherit its identifier from its generalization.
1.3 For each weak entity type in $M$, make the weak entity type derive its identifier from the strong entity types on which the weak entity type depends.

Step 2 Remove generalization hierarchies in $M$:
2.1 Remove every generalization hierarchy in $M$.
2.2 Generate a referential integrity constraint for each subset relationship in a generalization hierarchy in $M$.

Step 3 Check for the halting condition:
3.1 If every relationship in $M$ is marked with the word used, halt; otherwise, continue.

Step 4 Initialize a new ID scheme tree:
4.1 Select an unmarked relationship $R$ in $M$.
4.2 Create a single-branch ID scheme tree $T$ from $R$ such that each node in $T$ contains the identifier of a participating entity type of $R$.

4.3 Mark the root node of $T$ with the word continuation.
4.4 For each participating entity type $E$ of $R$ such that $E$'s (min-card, max-card) pair for $R$ is (1, 1), mark the node in $T$ that contains $E$'s identifier with the word continuation.
4.5 Mark $R$ with the word used.

Step 5 Extend an existing ID scheme tree:
5.1 If there is an unmarked relationship $R$ in $M$ and there is a participating entity type $E$ of $R$ such that we can find a node $N$ in an existing ID scheme tree $T$ that contains $E$'s identifier and $N$ is marked with the word continuation, continue; otherwise, go to Step 3.
5.2 Create a list $L$ from $R$-$E$ such that each node in $L$ contains the identifier of a participating entity type of $R$-$E$.
5.3 For each participating entity type $E'$ of $R$-$E$ such that $E'$'s (min-card, max-card) pair for $R$ is (1,1), mark the node in $L$ that contains $E'$'s identifier with the word continuation.
5.4 Update $T$ by attaching one end of $L$ as a child node to the node $N$ found in Step 5.1.
5.5 Mark $R$ with the word used and go back to Step 5.

Each ID scheme tree created by Procedure 1 is a skeleton. Procedure 2 adds attributes of relationships and entity types to the ID scheme trees produced by Procedure 1, which become the results of our scheme-tree generation algorithm. Note that after applying Procedure 1, every relationship in $M$ is marked with the word used but none of the entity types in $M$ has yet been marked. Procedure 2 will take advantage of this fact to ensure that it will only process each relationship and each entity type in $M$ once.

**Procedure 2.**

Input: an ID scheme-tree forest $F$
Output: a scheme-tree forest

Step 1 Add attributes of relationships to ID scheme trees:
1.1 If there is a relationship $R$ in $M$ that is marked with the word used, continue; otherwise go to Step 2.
1.2 If $R$ has attributes, let $X$ be the set of attributes of $R$; otherwise go to Step 1.6.
1.3 Let $T$ be the unique ID scheme tree in $F$ of which $R$ is a part.
1.4 Let $N$ be the lowest node in $T$ that contains the identifier of a participating entity type of $R$.

3 The notation $R$-$E$ means excluding the entity type $E$ from the relationship $R$. We create this notation since the relationship $R$ can be seen as a (multi)set of participating entity types.

4 Note that the identifiers of the participating entity types of $R$ form a consecutive sequence of nodes in $T$.  

---

<table>
<thead>
<tr>
<th>SSN</th>
<th>(SSN)*</th>
<th>(SSN)*</th>
</tr>
</thead>
</table>

Fig. 7. An scheme-tree instance over the ID scheme tree in Fig. 6.
1.5 Form a new node that contains all the single-valued attributes in \( X \) and add that new node as a child node to \( N \). Additionally, add each multivalued attribute in \( X \) as a separate child node to \( N \).

1.6 Unmark \( R \) and go back to Step 1.

Step 2 Add attributes of entity types to ID scheme trees:
2.1 If there is an unmarked entity type \( E \) in \( M \), continue; otherwise, halt.
2.2 If \( E \) has attributes, let \( X \) be the set of attributes of \( E \); otherwise go to Step 2.7.
2.3 If there is a node \( N \) in an ID scheme tree in \( F \) such that \( N \) contains \( E \)'s identifier and \( N \) is marked with the word continuation, continue; otherwise go to Step 2.6.
2.4 Add all the single-valued attributes in \( X \) to \( N \). Additionally, add each multivalued attribute in \( X \) as a separate child node to \( N \).
2.5 Go to Step 2.7.
2.6 Create a new scheme tree with all the single-valued attributes in \( X \) as the root node and each multivalued attribute in \( X \) as a separate child node to the root node.
2.7 Mark \( E \) with the word used and go back to Step 2.

3.2. The correctness of the algorithm

In this section, we prove that Procedures 1 and 2 together produce scheme trees that satisfy the four requirements we set forth at the beginning of Section 3. Since Step 2 of Procedure 1 obviously satisfies Requirement 3, we pay our attention to the other three requirements. We proceed with a series of lemmas. The first one is to prove that both Procedures 1 and 2 terminate.

Lemma 1. Procedure 1 and Procedure 2 both terminate.

Proof. We first focus on Procedure 1. Steps 1 and 2 both execute just once. Further, Steps 1.1, 1.2, 1.3, 2.1, and 2.2 clearly terminate. Thus, they do not concern us. Step 3.1 specifies the termination condition of Procedure 1 which we need to show that it will become true eventually. Step 4.1 selects an unmarked relationship. Steps 4.2, 4.3, and 4.4 each individually terminates. Step 4.5 marks it with the word used. Thus, the number of marked relationship increases by one. For Step 5, we have two cases to consider: (1) the condition of Step 5.1 is satisfied, and (2) the condition of Step 5.1 is not satisfied. In Case 1, since Steps 5.2, 5.3, and 5.4 each individually terminates, Step 5.5 will mark the unmarked relationship with the word used. In Case 2, Procedure 1 will go back to Step 3 and then Step 4 is executed again. Thus, in both cases, the number of marked relationship increases by one. Therefore, eventually the condition of Step 3.1 will become true and Procedure 1 terminates. Note that after Procedure 1, every relationship is marked with the word used. We now turn our attention to Procedure 2. Step 1.1 specifies the termination condition of Step 1. Step 1.2 identifies the set of attributes of a marked relationship. If that marked relationship does not have any attribute, Step 1.6 unmarks it and goes back to the beginning of Step 1. Otherwise, since Steps 1.3, 1.4, and 1.5 each individually terminates, Step 1.6 again unmarks the relationship and goes back to the beginning of Step 1. Thus, Step 1 of Procedure 2 terminates. For Step 2 of Procedure 2, Step 2.1 specifies its termination condition. Step 2.2 identifies the set of attributes of an unmarked entity type. There are two cases: (1) the unmarked entity type does not have attribute, and (2) the unmarked entity type has attributes. In Case 1, Step 2.7 immediately marks it and goes back to the beginning of Step 2. In Case 2, since Steps 2.3, 2.4, 2.5, and 2.6 each individually terminates, eventually Step 2.7 marks it and goes back to the beginning of Step 2. In either cases, the number of marked entity type increases by one. Thus, eventually the condition of Step 2.1 will become true and Step 2 of Procedure 2 terminates; after which, Procedure 2 terminates.

We next show that each instance of a relationship and each instance of an entity type in the input EER schema is stored in a unique nested tuple. In other words, we show that Requirements 1, 2 and 3 are satisfied.

Lemma 2. Each instance of a relationship in the input EER schema is stored in a unique nested tuple.

Proof. For this lemma, we only consider the scheme trees that are not produced by Step 2.6 of Procedure 2. We proceed by induction on the number \( n (n \geq 1) \) of relationships that appear in an output scheme tree \( T \). When \( n = 1 \), there is only one relationship \( R \) appeared in \( T \). Thus, \( T \) is created by Step 4 of Procedure 1 and \( R \)'s attributes are added to \( T \) by Step 1 of Procedure 2. Since \( T \) only contains \( R \) and no other relationships, each instance of \( R \) is clearly stored in a unique nested tuple and thus the basis of the induction is established. Assume this lemma is true if \( T \) contains \( k \) relationships for some \( k (k \geq 1) \). Now, consider adding the \((k + 1)\)-th relationship \( R \) to \( T \). Let \( E \) be the entity type of \( R \) such that there is a node \( N \) in \( T \) that contains \( E \)'s identifier and \( N \) is marked with the word continuation. By the induction hypothesis, each instance of each of the \( k \) relationships that appear in \( T \) is stored in a unique nested tuple. Since \( N \) is marked with the word continuation, there are two cases: (1) \( N \) is the root node of \( T \), and (2) \( E \)'s \((\text{min-card}, \text{max-card})\) pair must be \((1, 1)\) for some relationship that already appears in \( T \). In Case 1, by Definition 2 each instance of \( E \) appears only once. In Case 2, since \( E \)'s \((\text{min-card}, \text{max-card})\) pair is \((1, 1)\) for some relationship and by the induction hypothesis, each instance of that relationship only appears once, then each instance of \( E \) also appears only once. Since every instance of \( E \) appears only once, it is a complete join when we graft the \((k + 1)\)-th relationship \( R \) to \( T \) and thus each instance of \( R \) will also appear once. Hence, each instance of \( R \) is stored in a unique nested tuple. Finally, \( R \)'s attributes are added to \( T \) by Step 1.5 of Procedure 2. \( \Box \)
**Lemma 3.** Each instance of an entity type in the input EER schema is stored in a unique nested tuple.

**Proof.** In the proof for Lemma 2, we have showed that the instances of an entity type $E$ whose identifier is stored in a node $N$ that is marked with the word continuation appear only once. Finally, Step 2.4 of Procedure 2 adds all $E$’s attributes to $T$. Thus, these instances also appear in unique nested tuples. Step 2.6 of Procedure 2 creates a separate scheme tree for each entity type $E$ where $E$’s attributes appear only once. Finally, Step 2.4 of Procedure 2 adds instances of an entity type $E$ to $N$ all. Thus, these instances also appear in unique nested tuples. Obviously, these instances are stored in unique nested tuples as well. □

### 3.3 A case study

Example 7 shows how Procedure 1 constructs the ID scheme trees in Fig. 10 from the EER schema in Fig. 8.

**Example 7.** Fig. 8 shows a simple university EER schema. We now trace through the steps that Procedure 1 takes to create the left ID scheme tree in Fig. 10. Step 1 adds the identifier SSN to the entity types student and professor. Step 2 deletes the only generalization hierarchy in the EER schema, which results in the EER schema in Fig. 9. Step 2 also generates two referential integrity constraints to enforce that the social security numbers of the instances in professor and the social security numbers of the instances in student are both subsets of the social security numbers of the instances in person. Step 3 has no effect at this point. Suppose that Step 4.1 selects $E$’s and $E$’s attributes semester and grade as a child node to the node by student and planner. The node with the word continuation because planner’s (min-card, max-card) pair for belongs_to is (1,1). Finally, Step 4.5 marks belongs_to with the word used. At Step 5.1, the unmarked relationship advised_by satisfies the condition and therefore $R$ is advised_by and the node $N$ is SSN. Step 5.2 creates a list SSN, which in this case it has only a single-node. Since student’s (min-card, max-card) pair for advised_by is (1,1), Step 5.3 marks SSN with the word continuation. Step 5.4 adds student’s SSN as a child node to professor’s SSN. Step 5.5 marks advised_by with the word used and then Procedure 1 goes back to the beginning of Step 5. Eventually, the identifier title of the entity type course becomes two child nodes of student’s SSN because of the relationships enrolled and planned. However, none of them is marked with the word continuation since course’s (min-card, max-card) pair is (1,n) for both relationships. Procedure 1 creates another ID scheme tree from the other components in Fig. 9 and both ID scheme trees are shown in Fig. 10.

Example 8 demonstrates how Procedure 2 constructs the ID scheme trees in Fig. 11 from the ID scheme trees in Fig. 10.

**Example 8.** We now trace through the steps that Procedure 2 takes to create the scheme trees in Fig. 11. Step 1.5 adds enrolled’s attributes semester and grade as a child node to the node on the left and planner’s attribute semester as a child node to the node on the right. Consider the entity type department and its identifier name. The node that contains name has been marked with the word continuation and therefore Step 2.4 adds phone to it. For the entity type professor, the node that contains its identifier SSN has also been marked with the word continuation. Thus, Step 2.4 adds title to the node. Later, Step 2.4 does the same for the entity type student. Consider the entity type course and its multivalued attribute lab. The nodes

---

**Fig. 8.** A simple university EER schema.
that contain its identifier title are not marked with the word continuation. Therefore, Step 2.4 cannot add lab as a child node to anyone of them. Instead, Step 2.6 creates a new scheme tree with title as the root node and lab as a child node of title. The results are shown in Fig. 11.

4. Oracle Database 10g nested object types

Procedure 3, which is recursively defined below, maps scheme trees to Oracle Database 10g nested object types. Basically, Procedure 3 scans a scheme tree bottom-up from its leaf nodes to its root node.

Procedure 3.

Input: a scheme tree \( T \)
Output: a set of Oracle Database 10g create-object-type statements

Step 1 Leaf nodes:
1.1 For each leaf node \( N_i \) in \( T \), create an object type — denoted by type\( (N_i) \) — that includes each (single-valued) attribute in \( N_i \) as a field.

Step 2 Non-leaf nodes:
2.1 Let \( N \) be a non-leaf node in \( T \) and let \( N_1, N_2, \ldots, N_n \) \((n \geq 1)\) be the child nodes of \( N \).
2.2 Create an object type — denoted by type\( (N) \) — such that type\( (N) \) includes each (single-valued) attribute in \( N \) as a field.
2.3 If more than one instance of type\( (N_i) \) is allowed, type\( (N) \) has a field whose value is a collection of instances of type\( (N_i) \) \((1 \leq i \leq n)\).
2.4 If only one instance of type\( (N_i) \) is allowed, type\( (N) \) has a field whose value is a single instance of type\( e(N_i) \) \((1 \leq i \leq n)\).

Example 9 demonstrates how Procedure 3 generates the Oracle Database 10g create-object-type statements in Fig. 12 from the biggest scheme tree in Fig. 11. Note that all the create-object-type statements and the insert statement in Figs. 12 and 13 have been tested by Personal Oracle Database 10g Release 10.2.0.1.0.
create or replace type planned_attributes as object {
    semester varchar2(20),
    grade varchar2(2));
/
create or replace type planned_course_list as
    varray(200) of planned_course;
create or replace type planned_course as object {
    title varchar2(20),
    single_attributes enrolled_attributes);
create or replace type professor_list as
    varray(20) of professor;
create or replace type professor as object {
    name varchar2(20),
    ssn varchar2(9),
    varray(20) of student;
create or replace type student_list as
    varray(200) of student;
create or replace type student as object {
    name varchar2(20),
    ssn varchar2(9),
    varray(200) of planned_course;
create or replace type planned_course_list as
    varray(200) of planned_course;
create or replace type new_objects as object {
    name varchar2(20),
    phone varchar2(20),
    professors professor_list);
/}
create table departments of department;
/
create or replace type planned_attributes as object {
    semester varchar2(20));
/
create or replace type planned_course as object {
    title varchar2(20),
    single_attributes enrolled_attributes);
create or replace type professor as object {
    name varchar2(20),
    ssn varchar2(9),
    varray(20) of student;
create or replace type student_list as
    varray(200) of student;
create or replace type student as object {
    name varchar2(20),
    ssn varchar2(9),
    varray(200) of planned_course;
create or replace type planned_course_list as
    varray(200) of planned_course;
create or replace type planned_course as object {
    title varchar2(20),
    single_attributes enrolled_attributes);
create or replace type professor_list as
    varray(20) of professor;
create or replace type professor as object {
    name varchar2(20),
    phone varchar2(20),
    professors professor_list);
/
create table departments of department;
/
insert into departments values {
    department('MIS','(256)824-1111',professor_list{
        professor('111223333','Full',student_list{
            student('223344444','1',
                enrolled_course_list(,
                    enrolled_course('Database Management',enrolled_attributes('Fall 2002','A'))),
                    enrolled_course('Web Design',enrolled_attributes('Fall 2002','B'))),
                    enrolled_course('Basic Marketing',enrolled_attributes('Fall 2002','B'))),
                    planned_course_list(,
                        planned_course('Network Management',planned_attributes('Fall 2003'))),
                        planned_course('E-commerce',planned_attributes('Fall 2003'))),
            student('778899999','2',
                enrolled_course_list(,
                    enrolled_course('Database Management',enrolled_attributes('Fall 2001','C'))),
                    enrolled_course('Web Design',enrolled_attributes('Spring 2002','D'))),
                    planned_course_list(null))},
        professor('555667777','Associate',student_list(null))
})};
/
Fig. 12. Create-object-type statements.

Fig. 13. An insert statement.

Example 9. We now trace through the steps Procedure 3 takes to generate the create-object-type statements in Fig. 12 from the biggest scheme tree in Fig. 11. Step 1.1 generates two object types enrolled_attributes and planned_attributes from the two leaf nodes. Each of these two object types includes a field for each (single-valued) attribute contained in the corresponding leaf node. Step 2.2 then generates two object types enrolled_course and planned_course for the two nodes that contain title – the identifier of course. In addition, due to Step 2.4, enrolled_course also has a field whose type is enrolled_attributes because semester and grade are single-valued attributes of the relationship enrolled. Similarly, due to Step 2.4, planned_course also has a field whose type is planned_attributes because semester is a single-valued attribute of the relationship planned. However, the tree node that contains the identifier of student is different. Since a student may have enrolled in many courses and may have planned to take many courses, Step 2.3 needs to generate two collection types enrolled_course_list and planned_course_list. Hence, the object type student has two fields enrolled_courses and planned_courses whose types are, respectively, enrolled_course_list and planned_course_list. The object types for the nodes that contain the identifiers of professor and department can be generated similarly.

To illustrate how Oracle Database 10g stores permanent objects, the last statement in Fig. 12 actually creates a table that stores the instances of the object type department. To further illustrate how to use such a table, Fig. 13 shows an insert statement that inserts a department into the table. In this case, the department has two professors. The first one advises two students and the second one advises no students. Note that the referential integrity constraints produced by Procedure 1 can be implemented using object references (REFs) [21]. Since referential integrity constraints do not show the nesting structures of object types, they are omitted.
5. Conclusions

This paper focuses on the static aspects of object-relational databases. Particularly, we presented a methodology for designing proper user-defined type nesting structures for object-relational databases. Our methodology envisions that users model a real-world application with the EER model. Then, we apply some well-known techniques in the literature to obtain a semantically equivalent EER schema with no redundancy. Afterward, the redundancy-free EER model becomes the input to our scheme-tree generation algorithm. The result is a set of scheme trees that are both information content persevering and redundancy-free. Essentially, the central idea of our scheme-tree generation algorithm is to ensure that each instance of a relationship or an entity type is stored in a unique nested tuple. To guarantee such a property, our algorithm makes use of the (min-card, max-card) pairs of participating entity types in relationships. Finally, the output scheme trees are mapped to Oracle Database 10g nested object types for actual implementation. Nevertheless, this paper does not consider the dynamic aspects of object-relational databases. Particularly, we have not considered methods, triggers, functions and procedures of object-relational databases. We shall defer to a future paper for considering the dynamic aspects.

Acknowledgements

W.Y. Mok thanks the anonymous reviewers for their helpful suggestions. He was supported in part by the Richard A. Witmondt Faculty Fellowship and a UAH Research Mini-Grant.

References