THE DUAL LATTICE RELATIONAL DATA MODEL: AN APPROACH FOR MANAGING COMPOUND AND COMPLEX DATA TYPES

Hsiu-hsen Yao
Department of Computer Engineering and Science
Yuan-Ze Institute of Technology
Nei-li, Chungli, TaoYuan Shian
Taiwan

Abstract
In this paper, the dual lattice relational data model is proposed to combine the relational data model and the object-oriented data model. The dual lattice relational data model applies two different kinds of lattice structures for explicitly representing some specific relationships among relations. A brief description of the extended relational operators in the proposed model, such as LGEN, LPAR, MERGE, and SPLIT, is given. Finally, some kinds of translations from complex type data to compound type data are discussed.

Introduction
The relational data model proposed by Codd[1] is successfully used to manipulate commercial data, in which relations are chosen in first-normal-form (1NF) at least. The relations in 1NF allow queries to be composed directly (by using relational algebra), which provides a user-friendly interface between users and database management systems. Thus, the conventional relational data model is easy to understand, and then very popular.

However, the relational data model (with the first normal form limitation) has some defects:

1. The model is value-oriented, not object-oriented [2]. Thus, some features of object-oriented systems, such as inheritance of properties, set-valued types, and explicit representation of semantic meanings of data, are disappeared from the relational data model.

2. To maintain integrations of databases, the relational data model requires relation decompositions, which decreases system performance and make related information be separated in various relations.

3. It is difficult to maintain database integrities when users try to incorporate some changes of "objects" in real world into the database.

4. The concept of class is not represented in the relational data model explicitly. Some relations that belong to the same class are not bound together, which may be misunderstood by the users.

Ullman made mention of synthesis of object-oriented and value-oriented systems, and points out questions to the approach [2]. However, he did not give any description of features of these kind of compound systems.

In this paper, the dual lattice relational data model is proposed to combine the relational data model and the object-oriented data model. First, the assumption of first-normal-form relations is not necessary, while non-atomic attributes are allowed in relations. The new databases (with non-atomic attributes in relations) are named the non-first-normal-form (non1NF) databases. Some data storages and manipulations in CAD (computer aided design) or in SSDBS (statistical and science databases) may need non1NF databases since many "decomposable" data domains exist in such applications.

Makinouchi is one of the earliest researcher in non1NF relational databases [2]. He treats a multi-valued dependency (MVD) as a set-valued functional dependency (FD). Jaeschke and Schek then introduce the (one-level)NEST and (one-level)UNNEST operators to reorganize relations from 1NF to non1NF [4]. A nest operator groups sets of values in an attribute and an unnest operator takes sets apart.

Thomas and Fischer extend nest and unnest operators to allow multilevel nesting (and unnesting) on multivalued attributes [5]. The relations thus contain relations in nested attributes, which can be treated as tree-structured relations.

Yao and Yamashita then propose new relational operators which can be used to structure 1NF relations to lattice-structured relations[6,7]. A lattice-structured relation meets the needs of statistical analysis by subsetting data in the relation. The approach manages a great amount of data more efficiently without losing the merits of the relational data model. One the other hand, lattice structure is also used to organize classes in object-oriented systems. The purposed organization operators by Yao and Yamashita thus provide a possibility to combine the relational data model and the object-oriented data model. An object-oriented system (e.g., frame systems) can now be treated as views of a relational database and defined by some lattice structuring operators on some first-normal-form relations.
The operators NEST and UNNEST give us a way to "vertically" restructure data in a relation. On the other hand, we may "horizontally" organize data in a relation. That is, in a relational database, may happen that some relations have a scheme and are all partitioned ("horizontally") from an original larger relation. To maintain relationships among these subrelations, the conventional relational data model should be extended such that all related relations (with the same scheme) are linked together by directed arcs in lattices. The lattice structures are generated since two related relations may have a nonempty intersection.

Codd did some similar work in [8]. He developed an operator named FB R, denoted \( R \), which separates the set of records in a relation into different subrelations, where "every pair of these subrelations has an empty intersection" [8]. The operator FRAME may append to the relation a new attribute FID, frame identifier, which has some flavors of object-oriented data model.

In the proposed model, a non-atomic attribute generated by NEST and UNNEST operators may be set-valued, hierachy-structured, or lattice-structured. We say that the non-atomic attribute is in a compound data type. Comparing to the conventional relational data model, in which any database contains only one compound data type: the relation [8], the proposed model has more complex compound data types. These compound data types, however, can still be manipulated using (extended) relational operators, which does not increase complexities to the query and manipulative languages in the database management systems. Thus, when some applications of data processing need to handle "distinguishable" complex type data that can be "decomposed" into compound type data. Section 4 contains only one compound data type: the relation

A one-level nested relation is a relation with atomic and set-valued attributes. Two extended relational operators NEST and UNNEST, that change the nesting levels of nested relations, are given [4]. NEST groups atomic attributes into a set-valued attribute, and UNNEST decomposes a set-valued attribute into atomic attributes. For instance, consider a relation scheme \( R('Parent',Children) \). An instance \( r' \) with the scheme \( R \) contains two records: \((John, Steve)\) and \((John, Mary)\), which represents that John is the parent of both Steve and Mary. One may use the NEST operator to generate a (one-level) nested relation \( r'' \) with scheme \( R('Parent',Children) \), where \( r'' \) contains only one record \((John, \{Steve, Mary\})\). We say that the attribute Children has a compound data type. The user can get back \( r' \) from \( r'' \) by using the UNNEST operator. Since the set-valued attribute Children in \( r'' \) does not contain another sets recursively, the nested relation \( r'' \) is one-level.

The (one-level) NEST and UNNEST operators can be extended to the (multi-level) NEST and UNNEST operators [5], which allows the NEST operator to operate on some set-valued attributes. That is, the nested relation is hierarchy-structured. The (multi-level) UNNEST operator is the inverse operator of the (multi-level) NEST operator, which gets the original relation back. For instance, consider a organization relation \( r \) with the scheme \( R('Division', Dept, Prof) \). Firstly, we nest the attribute Prof to get a nested relation \( r' \) with the scheme \( R('Division', Dept, Profs) \), where the attribute Profs contains all professors in any given division and department. Secondly, we nest the attributes of Dept and Profs in the relation \( r' \) and then get a hierarchy-structured relation \( r'' \) with scheme \( R('Division', Dept, Profs) \), where the attribute \((Dept, Profs) \) is a (multi-level) nested attribute.

To extend multi-level nested relations to lattice nested relations, we briefly describe the concept of lattices. A lattice, that can be viewed as an extension of a tree, has a common form of hierarchy (i.e., an acyclic graph), where a partial ordering is set on. Each node in the lattice may have more than one "parent node", and every pair of nodes must have only one "minimal ancestor node" and one "maximal descendant node". Consider a relation \( r \) with the scheme \( R('ABC') \), where \( C \) is a set-valued attribute, if two NEST operators are allowed to nest in the relation \( r \) on the attributes \( BC \) and \( AC \) (i.e., two nested attributes have an intersection), then the result relation is not tree-structured but lattice-structured. This kind of nesting operators are called the (lattice) NEST operators [6].

We may define the LGEN operators instead of using the (lattice) NEST and UNNEST operators. Let \( LGEN[Li](r) \) represents that the relation \( r \) has been (lattice) nested (or unnested) into \( r' \), where \( L \) is the assigned lattice structure, and \( r \) is a first-normal-form relation.

The (one-level) NEST and (one-level) UNNEST operators, and the (multi-level) NEST and (multi-level) UNNEST operators are now special operators of LGEN since a tree can be viewed as a kind of lattice. Instead of decomposing a large first-normal-form relation into several smaller higher level, say 4NF, relations (i.e., 1 normalization), we may lattice-generate the relation. Since all related data are still in a relation, which prevents join operations in queries, execution times of query processors can be reduced. In addition, lattice-structured relations store data with minimal duplications of data items.
In real world applications of data processing, the cycle of data processing can be divided into three phases, i.e., ETL, input data first, then process the data, and output the result finally. One may find that the input data sheets in Y and the output reports in Z can all be viewed as non-first-normal-form relations. Thus, the database relations, that are used as data storage files in P, should also be in non-1NF.

Note that each nested attribute can be viewed as a small relation within a "bigger" relation, we give these "small" relations a name: the relation units.

We now consider a two-layer structured relation r', which has been (lattice)nested already. r' can be viewed as a storage object that contains two parts of data structures: r and s, denoted <r, s>, where r represents the 1NF relational scheme of r' and s is the lattice structure of r'. The lattice relational algebra can be defined as follows:

The operators of UNION, SELECT, PROJECT, JOIN, and PROJECT, are conventional 1NF relational operators.

1) union: \( \text{UNION}(r', r') = \text{LGEN}(\text{UNION}(r, r)) \), where r and R are two 1NF relations of r' and r, respectively; and s is the lattice structure of r'.

2) select: \( \text{SELECT}(r'; r') = \text{LGEN}(\text{SELECT}(r', r)) \), where \( F \) is given to filter r', the corresponding condition of \( F' \).

3) join: \( \text{JOIN}(s'; r', r') = \text{LGEN}(\text{JOIN}(r', r')) \), where \( s' \) is a lattice structure given by the user.

4) project: \( \text{PROJECT}(Yr; r') = \text{LGEN}(\text{PROJECT}(Yr)) \), where \( s' \) is a given lattice and Y is the attributes in r, and r is the 1NF relation of r'.

The operators of SELECT, UNION, JOIN, and PROJECT, are the lattice relational operators. The rest of the lattice relational operators can be defined similarly by operating on 1NF schemes first and then nesting the result 1NF relation into the (lattice)nested relation.

The Dual Lattice Relational Data Model

Consider a set of first-normal-form relations with the same scheme. Operators such as union, difference, intersection, and selection, may be used to create queries (and thus views) with the same scheme. Relationships among them generate a lattice structure if we consider relation union and intersection to be two binary operations Lattice-Join and Lattice-Meet, respectively [9]. To maintain database integrity, the system needs to know some meta-data about the relationships among relations with the same scheme.

As we discuss in Section 2, a database contains a set of relation units. Each relation unit with its descendant relation units can generate a relation. Each relation in the lattice relational data model has one and only one 1NF representation. Consider a set of relations, denoted R, with the same 1NF scheme (i.e., after unnesting the relations in R, the corresponding first-normal-form relations, they all have the same scheme). R is a component if there is no relation r such that r has the same 1NF scheme as the relations in R, and r is not in R. We now define an operator LPAR (lattice-partition), which generates a lattice to "partition" any relation r in R. Let L be a lattice, in which each node in L is either r or a view generated by using union, difference, and/or selection operators on r.

LPAR(L)(r) contains a set of relation pieces, where each relation piece is corresponding with a node in L. In other words, the LPAR operator adds a new lattice structure into r.

Let r' be a union of relations in r. The user can access the materials relations in the database by executing STORE(LPAR(L)(r')). If a user executes STORE(LPAR(L)(r')), where r' is not equal to r, then the system gives a new lattice L' into the database such that L is a subgraph of L' (i.e., LPAR(L)(r) = LPAR(L)(r')).

It is, thus, very difficult to design the lattice relational algebra on the proposed model. The dual lattice relational algebra contains LGEN, LPAR, and some extended operators such as select, PROJECT, JOIN, DIFFERENCE, and UNION. A two layer structure is assigned to any relations, where the dual lattice structure is in the second layer, and the 1NF scheme is in the first layer. The operators are now executed to operate the related first-normal-form relations firstly, then to manage the dual lattice structures secondly. The algebra is similar to the lattice relational algebra. The proposed algebra remains the merits of the relational data model.

The purposes of designing the dual lattice relational data model are that:

1) the database designers generate a relation as large as possible. Then, the designers split relations into relation units and relation pieces; and

2) the users retrieve data from the database by using the concepts of the relational data model and/or the object-oriented data model.

We now apply the concept of semantic networks to explain the proposed model:

Consider a relation r, where its primary key is X and an attribute Y is in a compound data type.

1) Y is generated by LGEN, which makes the link of X to Y have semantic meanings of IS_A, ARO, or IS_A_PART_OF.

Let r' be a relation piece of r.

2) r' is generated by LPAR from r, which makes the link of r to r' have semantic meanings of SUBSET_OF.

In object-oriented systems, classes being used to classify different types of objects, are organized to a lattice structure, which is similar to the lattice structure of relation units in the dual lattice relational data model. The proposed model allows various related data (e.g., row data and summary data in a SSDB) merged into one relation but designed into different relation units, that makes data manipulations very convenient in applications. Thus, the property of inheritance can be handled in the dual lattice relational data model.
The difference between the concepts of object-oriented systems and the proposed model is that, in object-oriented systems, each object has an identity created by the system, but the dual lattice relational data model allows using primary key instead of object-id, which may be more reasonable in a real world.

Based on some analysis of features of data in a database, we can classify them into three different cases. Some data are only being-retrieved in a relation, and the others are either a primary key or category attributes. For instance, in a science database, most of experimental data are only being-retrieved; and in a statistical database, summary data play the same role, but category attributes may be used in some access paths. In the dual lattice relational data model, a lot of queries can follow the lattice paths to get data.

Complex Data Types and Compound Data Types

As we discuss in Section 2, the compound data type is a type generated from atomic data types by the operator LGEN. In this paper, we use compound data types to represent structured relations such as set-valued, hierarchy-structured, or lattice-structured attributes. Any attribute in a compound data type can always be decomposed (i.e., unwrapped) into a set of attributes in original atomic types.

There are a lot of applications that needs to manage "distinguishable" complex type data. Some examples of these kinds of complex type data are listed here:

1. **TIME type**: a value with the type of HH:MM:SS, where 0 <= HH <= 24, (hour), 0 <= MM <= 60, (minute), 0 <= SS <= 60, (second);
2. **DATE type**: a value with the type of MM:DD:YY, where 1 <= MM <= 12, (month), 1 <= DD <= 31, (day), 1 <= YY <= 99, (year);
3. **GRAPH type**: a finite graph G = (V, E), where V is a finite set of nodes, and E is the set of arcs that connect pairs of nodes; the TREE type and the Lattice type are special types of the GRAPH type;
4. **TEXT type**: a TEXT type data (i.e., a string) can be viewed as a sequence of characters, or a sequence of tokens;
5. **Hypertext type**: a HyperTEXT type data can be viewed as a finite arc labeled graph G = (V, E'), where each node in V is in TEXT type;
6. **LIST type and MATRIX type**: a list (L[i,1], i), 1 = 1 .. N, is an ordered set; and a matrix (M[i,1], 1, j), i = 1 .. M and j = 1 .. K, can be viewed as a two dimensional list.

Data of these six complex types above can all be decomposed into smaller data items. To manage these complex types, most of database management systems provide build-in functions to support domains of these complex types. For instance, to support the DATE type, a database management system may need to provide functions such as time zone independence, a special function NOW, arithmetic functions on dates, or comparison of dates.

Since the system may not provide any functions on complex data types which satisfy all users need, it is reasonable to translate complex data types into the corresponding compound data types, and the users can manipulate these decomposed data items by relational operators and then get the data they need. Using relational operators to retrieve data is more flexible and, thus, more powerful. The translations may eliminate some features of the complex type data, but the values of the components in a complex type data can still be in the relation. Note that translatable data types such as list, tree, graph, network, or matrix, should be the types having distinguishable components.

We now define two relational operators which have polymorphism.

1. The operator SPLIT(A)^r, where r is a relation, and A is an attribute in the complex data types that are given above. The result relation contains a compound type attribute:
   - (i) if A is in DATE type or in TIME type then SPLIT may separate A into three attributes MM, DD, YY, (i.e., data of month, day, and year), or HH, MM, SS, (i.e., data of hour, minute, and second), respectively;
   - (ii) if A is in GRAPH type then SPLIT may generate a subrelation (i.e., a compound data type) which consists of two attributes V and E, where V contains values of the nodes and E is a compound type that consists of a set of adjacent nodes of the node in V. When the type is arc labeled GRAPH type, the attribute E should also consist of the third attribute named Label;
   - (iii) if A is in LIST type then SPLIT may create a compound type that consists of two attributes I and L, where L contains component data of the list in A, and I represents the index of L;
2. The operator MERGE[A]^T^r, where A is a compound type attribute name, T is a complex data type, and r is a relation. The type T should be supported by the system and any execution of the operator MERGE will check domain integrity of T.

Note that some set-valued type may not be a compound data type [10]. To get the compounds of these sets, the system may support relation-id (just like object-identifier) to the users. When the users need this information, they can use GEN(r) to get it which stores in an attribute (i.e., the users can see it). The relation-id can be assigned in any compound type attributes (and relations).

The dual lattice relational data model can be used in a lot of applications such as meta-data retrieval, version management, exception handle, and time interval type management. As long as we define a new type for a new applications, the most important domain function is the translation (or even just a mapping) between the type and the corresponding compound data type, which provides the users a flexible and powerful tool to use relational operators to manipulate or to retrieve data.

738
Conclusions and Further Works

In this paper, we briefly give a description of the dual lattice relational data model. The model adds two different kinds of lattices into relations and the proposed (extended) relational operators retain the merits of the conventional relational algebra (i.e., there is only one type of a result file: the relation, which makes compositions easily). Finally, we discuss some translation operators to change a complex type data into a compound type data which allow the users to manipulate some components of a complex type data only applying the relational operators.

A database system design is now different from the conventional database design. In the dual lattice relational database system, the database designers are encouraged to "link" related data in a relation, and decompositions of relations become unnecessary. The design process needs new guides to follow. The proposed model also gives an approach to combine both the database system and the knowledgebase system, which need more researchs on the proposed model.

Reference