MODELING AND QUERYING RECURSIVE DATA STRUCTURES II: A SEMANTIC APPROACH

J.H. ter Bekke and J.A. Bakker
Delft University of Technology
Faculty of Information Technology and Systems
e-mail: {J.H.terBekke, J.A.Bakker}@its.tudelft.nl

ABSTRACT

As shown in the introductory paper [7], the semantic modeling approach enables us to specify relationships in an inherent way. Consequently, designers can specify recursive structures in a way complying with the recursive mathematical definition of a series of variables. For end users this has the advantage that they can ignore procedural aspects in query specifications for recursion; the required processing details are derived by intelligent software able to interpret the underlying inherently structured metadata, which results in a reliable (finite) and efficient processing. We demonstrate this by examples of recursive and non-recursive queries using a recursive data model for family trees, which is implemented in a working database management system.

KEYWORDS

Query language, recursive query processing, semantic data modeling, metadata, end user computing, reachability, transitive closure, expressive power.

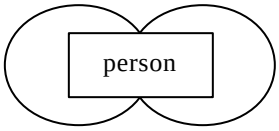
1. Introduction

As stated in the preceding paper [7], the definition of a collection of objects O_1, O_2, ... is recursive when O_1 is defined on known concepts and the definition of O_{n+1} is based on the definition of O_n for positive n: the predecessor occurs as a property (or attribute) in the definition of each successor. The semantic approach to data modeling complies with this definition of recursion: depending on the context, also in the case of recursive data models, objects can be interpreted either as 'type' or as 'attribute'. This is essential for the specification of recursive queries by end users; they still can apply the *its* construct. We confine the discussion to applications for recursive data structures and require that both recursive and non-recursive applications are specified declaratively, thus the application of a declarative language without explicit recursion, nesting, iteration and navigation. This absence of procedural details is crucial for end user computing. Such an approach requires that the software of a database management system is able to determine the required processing details from the given query specification in a reliable and efficient way.

After a short introduction to the required semantic concepts in section 2, some queries will be specified using the recursive data model for family trees in section 3. All used semantic concepts [1, 2] have been implemented and extensively tested using the Xplain DBMS, version 5.7 [3]. Therefore section 5 contains some implementation considerations.

2. Semantic Concepts

Each object in a semantic model is visualized explicitly by clearly distinguishing between identification and descriptive properties, which avoids ambiguities and contradictions in specifications. Only three fundamental abstractions with clear graphical equivalencies in the structural diagrams are required to specify semantic integrity in an inherent way. These abstractions apply the fundamental type-attribute relationship. A first abstraction is classification: to categorize objects perceived in some relevant world into types. A type (identified by a name) is a fundamental concept; it can be represented by a rectangle in diagrams as in figure 1.



type person = from_year, to_year,
mother_person, father_person.

Figure 1. Recursive aggregation

A second abstraction is aggregation: this is defined as the collection of a certain number of types in a unit, which in itself can be regarded as a new type. A type occurring in an aggregation is called an attribute of the new type. Note the analogy with the mathematical set concept: attributes are considered as the 'elements' of a type. Aggregation allows view independence: we can discuss the obtained

type (possibly as a property) without referring to the underlying attributes. By applying this principle repeatedly, a hierarchy of types can be set up, for example the recursive hierarchy for a family tree in figure 1. Normally such a diagram only shows composite types; types constructed by aggregation. Base types are not based on aggregation and are not shown in such a diagram. A line connecting two facing rectangle sides, while the aggregate type is placed above its attributes, indicates aggregation. Aggregation has its counterpart: the description of a type as a set of attributes is called decomposition. Because of its definition aggregation does not require a directed arc in the diagram. As will be illustrated later, the aggregation relationship can be applied different directions. A type is completely defined by listing its attributes, so we could define the type 'person' as in figure 1, which is the basis for the registration of family trees. This definition is in the earlier given general sense of recursion a proper recursive definition because a person is defined in terms of its two predecessors, viz. 'father_person' and 'mother_person'. Both attributes refer to the type 'person'; the type definition contains therefore an illustration of the concept of object relativity, saying that type and attribute are different interpretations of the same object. The interpretation is determined by the context. For example with '*get* person' we retrieve all instances of 'person', whereas with '*get* person "Jan" *its* mother_person' we select a specific attribute value. A part of the family database is given in table 1.

person	from_ year	to_ year	father_ person	mother_ person
?	?	?	?	?
..	..	..	..	..
Karen	1809	1892	William	Ellen
Joseph	1811	1892	?	Anne
Petra	1838	1912	Joseph	Karen
Irene	1839	1899	Joseph	Karen
Margaret	1843	1923	Joseph	Karen
..	..	..	..	..

Table 1. Part of a family tree database

Type definitions carry semantics; they contain the essential properties (e.g. uniqueness of the identifications 'Karen', 'Joseph', etc. and essential relationships (e.g. related persons 'William', 'Bernhard', 'Ellen', etc. must occur in the related person instances). Aggregation can be described using the verb to have. For example, each instance of 'person' has a 'from_year', a 'to_year', a 'father_person' and a 'mother_person'. Identifications are properties denoted by type names (see table 1 above). This interpretation implies singular identifications for instances (implementation level) and types (conceptual level). Attributes (not types!) may contain roles; examples are 'from_year' and 'to_year', related to base type 'year'. The attributes 'father_person' and 'mother_person' are related to the type 'person'. Roles are separated from the type by an underscore. The expression 'A *its* B' denotes the attribute B of type A.

Because the recursive data model in figure 1 consists of only one composite type, the starting point for recursion (the root in the earlier definition of recursion) must be given first. In this example we use the default value "?" for this purpose. Applied to a person this root instance has itself as father and as mother. As instance identification this value occurs only once. As attribute value this value may occur several times (also as a domain value for other unrelated attributes). This default value solution allows us to use two-valued logic in query language statements. The value ".." is used because only a part of the family tree is given in table 1. It is used to denote places where defined instance values occur. As a result all earlier shortcoming of a nested structure are disappeared:

- All persons (both parent and child) are modeled in the same way. The desired properties (e.g. being parent or child) can be derived easily.
- Completeness and consistency is no issue anymore. Each person has exactly two parents. When a parent is unknown, this is derivable from the presence of a certain attribute value as "?", but not from the absence of a value.
- The structure contains no undesirable asymmetry. It is simple to determine the children of a certain person. It is also simple to determine the parents of a certain person.
- The essential relationship between persons is present: from the inherently defined structure it is clear that a person always has two parents.
- Simple update properties: The fact that a person becomes parent must be recorded only once. Children with the same parent are easily to derive.
- The structure contains no redundancy: each fact is recorded exactly once.

3. Applications

Applications on a recursive data structure can have a recursive or a non-recursive character. This section presents a number of examples to illustrate different

interpretations of the same semantic relationship. These interpretations require different language elements. The elements are fixed in the Xplain data model; they are derived from the inherent constraints in a semantic data model. They prevent pitfalls and misinterpretations and result also in unique query specifications. A query consists always of a limited number of semantic definitions (derivations) and a selection. The first query is an example of the *extend* command. This command does not require any processing order.

Query 1: Determine children

extend person *with* father = (1.1)
    any person (1.2)
    per father_person. (1.3)
extend person *with* mother = (1.4)
    any person (1.5)
    per mother_person. (1.6)
get person (1.7)
    where not father *and not* mother. (1.8)

Brief explanation:
(1.1)..(1.3) We start the query specification with a definition of a father. A father is a person occurring in at least one instance as 'father_person'; this new property is recorded as a Boolean attribute. A defined extension can be conceived as a normal attribute, but disappears after query execution.
(1.4)..(1.6) Analogously the property mother is defined as a Boolean attribute.
(1.7)..(1.8) Child is a person not being a father and not being a mother.

Prelude to related persons
Following queries 2 to 6 have to do with persons related in one or another way to a particular person. For example, we are interested in brothers and sisters, cousins, descendants and ancestors of a particular person. In all cases the user determines the relevant person. For this the following fixed construct is used in the query specification:

value name = *input*(a15) 'Enter name:'. (0.1)
extend person *with* himself = (*false*). (0.2)
update person name *its* himself = (*true*). (0.3)

Brief explanation:
(0.1) User is asked to give the name of the relevant person. The prompt consists of the text 'Enter name:'. A name may contain at most 15 alphanumeric characters.
(0.2) Initially for all persons a new attribute 'himself' is defined. This attribute is given the value false, denoting that the person is not the relevant person in question.
(0.3) Only the person with the given name receives a true value. This *update* command enables us to select the person identified with name from the collection of persons. The *update* command is also used in the following queries in order to exclude the root of the family tree.

The value for the extended attribute 'himself' is determined in all queries in the same way. That is the reason why the steps (0.1)..(0.3) are not repeated in the following queries. We start with some non-recursive applications. The required information can be derived from the database by using *extend* commands. These simple applications are dealing with a fixed number of levels in the family tree. Examples are:

- Determine the brothers and sisters of a person.
- Determine the cousins of a person.

Brothers and sisters have the same father. Cousins have the same grandfather, but not the same father. The first queries require first definitions of persons with these properties. Then, persons with a special relationship can be selected from the collection of persons.

Query 2: Determine brothers and sisters

extend person *with* ownfather = (2.1)
    any person *where* himself (2.2)
    per father_person. (2.3)
update person "?" *its* ownfather = (*false*). (2.4)
get person (2.5)
        where not himself
        and father_person *its* ownfather. (2.6)

Brief explanation:
(2.1) The type 'person' is extended with attribute 'ownfather' denoting whether the person is father of 'himself'.
(2.2) Function *any* results in a logical value. The 'ownfather' has value true; others have value false.
(2.3) The 'ownfather' has a relationship with the relevant person. Grouping is done on the basis of attribute 'father_person'. Persons who are not father will have the value false.
(2.4) The root of the family tree must be excluded. When the root is not excluded, all persons with unknown father will be selected.
(2.5) Because no attributes are specified, by default all attributes of a person are selected.
(2.6) Select persons who satisfy the condition that the person has the same father, 'himself' excluded. Join terms are not needed because necessary metadata is available.

Query 3: Determine cousins

Cousins of a certain person are those persons having the same grandfather but not the same father. Each person has two grandfathers, one via the mother and the other via the father. We need two similar extends:

```
extend person with grandfather1 =                    (3.1)
    any person where himself                         (3.2)
    per father_person its father_person.             (3.3)
update person "?" its grandfather1 = (false)         (3.4)
extend person with grandfather2 =                    (3.5)
any person where himself                             (3.6)
    per mother_person its father_person.             (3.7)
update person "?" its grandfather2 = (false)         (3.8)
extend person with ownfather =                       (3.9)
    any person where himself                         (3.10)
    per father_person.                               (3.11)
update person "?" its ownfather = (false)            (3.12)
get person                                           (3.13)
  where not father_person its ownfather              (3.14)
  and (father_person its father_person
                its grandfather1                     (3.15)
  or mother_person its father_person
                its grandfather2).                   (3.16)
```

Brief explanation:
(3.1)..(3.4) We determine the first grandfather via the person's father.
(3.5)..(3.8) The second grandfather is determined.
(3.9)..(3.12) The father is determined to be able to exclude brothers, sisters and 'himself'.
(3.13)..(3.16) Cousins do not have the same father and have one of the two grandfathers.

Foregoing solutions have all the same structure: a number of derivations followed by a selection. Recursive queries consist (according the definition of recursion) always of at least two steps: initialization and *cascade* (the recurrent term). The recurrent term is completely driven by the database contents. The value of a recurrent term can change during execution. An *extend*-command can therefore not be applied: it is not possible to use a value for modification while its definition is not completed (or even not exists). That is the reason why all recursive queries contain the *cascade* update command. In this command both directions present in a relationship can be used.

The definite relationship requires the *its* construct (for example: '*get* person *its* mother_person'), whereas the variable relationship (for example: '*count* person *per* mother_person') requires a set function and the *per* construct.

Query 4: Determine descendants

Descendants of a certain person have a father or mother who is also a descendant. The start of the recursion is the person 'himself'. Next the *cascade* is executed. A person is a descendant if it concerns 'himself' or if father or mother is already a descendant. The necessary ordering (first father and mother) is determined during query parsing and is immediately used in query execution. This ordering is determined by the software system (in our case Lex and Yacc) using the metadata.

```
extend person with descendant = (false).             (4.1)
update person name its descendant = (true).          (4.2)
cascade person its descendant =                      (4.3)
    (descendant                                      (4.4)
     or father_person its descendant                 (4.5)
     or mother_person its descendant ).              (4.6)
get person its from_year                             (4.7)
    where not himself and descendant                 (4.8)
    per from_year.                                   (4.9)
```

Brief explanation:
(4.1)..(4.2) Person 'himself' is the starting point for the recursive process.
(4.3)..(4.6) The *cascade* command contains the recurrent term. It is important that father and mother are marked as a descendant before the person can be marked as a descendant. The root cannot be a descendant.
(4.7)..(4.9) Descendants, 'himself' excluded, are selected. The result is given in ascending 'from_year' order.

The definite relationship can also be used in the reverse direction. In that case a set function is needed together with the *per* construct. Example queries concern the determination of ancestors and forefathers. Again a *cascade* follows the initialization. Different grouping criteria can be used.

Query 5: Determine ancestors

```
extend person with ancestor = (false).               (5.1)
update person name its ancestor = (true).            (5.2)
cascade person its ancestor =                        (5.3)
    any person where ancestor                        (5.4)
    per father_person, mother_person.                (5.5)
update person "?" its ancestor = (false).            (5.6)
get person its from_year                             (5.7)
    where not himself and ancestor                   (5.8)
    per from_year.                                   (5.9)
```

Brief explanation:
(5.1)..(5.2) The person with the given name is starting point for the *cascade* command.
(5.3)..(5.5) In the recurrent term for the determination of ancestors, fathers and mothers do play a role. When a person is already ancestor, then father and mother is also ancestor.
(5.6)..(5.9) The root of the family tree is excluded. Select all ancestors and exclude the person 'himself'. The result is given in ascending order of 'from_year' ('*per* from_year').

Query 6: Determine forefathers.

In this query we are interested in the direct ancestor line in the family tree, that is the line: person 'himself', father (person's father), grandfather (father's father), etc.

extend person *with* forefather = (*false*). (6.1)
update person name *its* forefather = (*true*). (6.2)
cascade person *its* forefather = (6.3)
    any person *where* forefather (6.4)
    per father_person. (6.5)
update person "?" *its* forefather = (*false*). (6.6)
get person *its* from_year (6.7)
    where not himself *and* forefather (6.8)
    per from_year. (6.9)

Brief explanation:
(6.1)..(6.2) The starting point for recursion is given.
(6.3)..(6.5) The father plays a role in the recurrent term.
(6.6)..(6.9) The root of the family tree is excluded. Select all forefathers, the root excluded. The result is given in ascending order of 'from_year' ('*per* from_year').

Query 7: Determine number of generations.

extend person *with* generation = -1. (7.1)
cascade person *its* generation = (7.2)
max (father_person *its* generation,
    mother_person *its* generation) + 1 (7.3)
get max person *its* generation. (7.4)

Brief explanation:
(7.1) The root of the family tree has 'himself' as a father and mother: the generation should initially start with -1. The *cascade* command will set this value to 0.
(7.2)..(7.3) A child from a father and/or a mother will be a person of the next generation. If only the direct descendants line is needed the maximum of father's and mother's generation should be replaced by the attribute: 'father_person *its* generation'.
(7.4) The maximum value for generation is selected.

The last example illustrates that not only logical operations but also computations can be specified by *cascade* commands.

4. Consequences

In the presented unique approach both designer and end user do not specify nor use navigation through data; they only follow path's existing in the model. This absence of procedural details in their specifications requires that the responsibility for process ordering be delegated solely to the software system.

Because only definite relationships may be used, the system must take this responsibility using the underlying data model. This separation of concerns has been implemented in the Xplain software system version 5.7 and has two important consequences:

- Ordering is not specified
  Designers don't have to tune their design to the processing order desired by end users: ordering can be used on the fly without any warning.
- Errors are reduced
  Incorrect or incomplete process specification is not an issue anymore, which is important for both designer and user. The software guarantees finiteness of processing.

It took us much time before we clearly understood that the required procedural details (ordering, guaranteeing termination) for recursive processing could be inferred from declarative query specifications. Although this was already realized for non-recursive queries (also in relational systems), the Xplain approach has given a fundamental extension to the expressive power of a declarative query language. Advantages of this new approach are gigantic:

- A possible recursive ordering has no consequences for storage and access to data. Recursive processes can be specified on the fly, without procedural details.
- The correct and optimal processing is in all cases determined by software. This implies a gain of time for all database users.
- Processes are reliable because of a minor dependency of user and designer. Only the software guarantees the finiteness of applications. This is important in environments where several users are using the same database. This becomes even more important when databases are accessible by an enormous number of unknown users through the Internet; then it is impossible to protect systems by authorization tables.

5. Implementation

We can distinguish two forms of the *cascade* update command: the first form is based the use of on a set function and the other form without set function. The first form uses the variable (derivable) relationship and the second form uses the specified definite relationship. The general structure of these commands is as follows:

1. *cascade* <subtype> *its* <cascade attribute>
   = <function> <maintype> *its* <expression>
   per <group>.

2. *cascade* <subtype> *its* <cascade attribute>
   = <expression>.

In case of a recursive data structure both <subtype> and <maintype> indicate the same type from the data model. The other difference is that <group> may consist of two attributes ('*per* father_person, mother_person') instead of

only one attribute. The <group> attributes must be related to <subtype>. Also this is easily determined during query parsing.

The required ordering is defined by the instance related to the <cascade attribute> in <expression> and the <cascade attribute> in the instance of <maintype> related to <group>. There is only a recursive process if both <expression> and <group> are related to different roles of the type with the <cascade attribute> of <maintype>. A processing order is determined such that <expression> of an instance is known before the value of <cascade attribute> is assigned. During query preparation the recursive ordering of instances in <maintype> is determined using the different roles.

The second form of the *cascade* command is unique for recursive data structures. There is only a recursive process if <expression> contains the <cascade attribute>. The related instance is determined by using the different roles. Query processing is such that the values in <expression> are known before the result is assigned to <cascade attribute>.

An overview of the foregoing recursive queries is given in table 2; each step in a recursive process uses the value of a cascade attribute belonging to an originating instance (of 'person') and assigns a calculated value to a destined instance (of 'person').

The ordering of these successive assignments (source and destination) is inferred from the query specification. The query language processor determines all metadata required for query execution. This is done during parsing time of the query. In our case Lex and Yacc determine the required recursive ordering.

query	cascade attribute	source	destination
4	person *its* descendant	father_person, mother_person	person
5	person *its* ancester	person	father_person, mother_person
6	person *its* forefather	person	father_person
7	person *its* generation	father_person, mother_person	person

Table 2. Cascade attributes and direction of recursive operations

Query execution consists of the following steps:

1. Determine the recursive ordering for the processing of instances of <maintype>. Present a clear error message if this ordering cannot be determined.
2. Execute the command as an *update* command according the ordering from step 1.

A description of the implementation of step 1 in Xplain DBMS (including error handling) has been given in [6]. Normally the processing of a query command consists only of step 2, in which a system defined ordering is used. This is also the ordering for the *update* command in case the *cascade* command is used without fulfilling the required preconditions.

The performance of this declarative solution for recursive queries is extremely high. Processing of recursive queries (including parsing, execution and presentation of the result) is within almost linear processing time. This was already confirmed by extensive tests on small, medium and large databases [4, 5]. The queries in this paper have only been tested on a small database concerning the family tree of a royal family as found in literature consisting of 80 persons and 10 generations. The elapsed time for previous queries was always within 0.06 sec. on a Pentium II notebook under standard Linux.

6. Conclusion

The semantic model can be used to represent important mathematical structures as graphs and recursive data structures. The consequence of inherent representation of definite relationships is that both recursive and non-recursive applications can be described using a declarative query language. This means that the semantic model is more than relational complete.
Designers and users don't have to specify procedural details; all required semantic information is derived on the fly from inherently structured data by a query language processor.

The emphasis of other approaches on variable relationships has caused blockades on several fronts. View modeling (modeling a singular application) instead of conceptual modeling makes it difficult to derive other views. Furthermore, recursive applications must be specified in complex computer programs (including recursion, nesting, navigation and iteration) instead of declarative queries. These shortcomings hold for hierarchical, network and object-oriented models.

Exceptions are the relational and the semantic model. Both emphasize definite relationships, but only the semantic model endorses an inherent specification of definite relationships. The resulting exchangeability of type and attribute enables us to use the semantic model for declarative specifications of recursive queries, also on recursive data structures.

References

[1] F. Rolland, *The essence of databases*, Prentice Hall, Hemel Hempstead, 1998.

[2] J.H. ter Bekke, *Semantic Data Modeling*, Prentice Hall, Hemel Hempstead, 1992.

[3] J.H. ter Bekke, Advantages of a compact semantic meta model, *Proceedings 2nd IEEE Metadata Conference*, Silver Spring (1997). http://www.computer.org/conferen/proceed/meta97/papers/jterbekke/jterbekke.html.

[4] J.H. ter Bekke and J.A. Bakker, Recursive queries in product databases, *Proceedings 5th International Conference on Flexible Query Answering Systems* (FQAS 2002), Copenhagen, Denmark, October 27-29, 2002, Lecture Notes in Computer Science (Subseries LNAI) Vol. 2522, T. Andreasen, A. Motro, H. Christiansen, H. Legind Larsen (Eds.), Springer-Verlag, Berlin-Heidelberg (2002), pp. 44-55.

[5] J.H. ter Bekke and J.A. Bakker, Content-driven specifications for recursive project planning applications, *Proceedings International Conference on Applied Informatics* (AI 2002), Innsbruck, Austria (2002), ed. M.H. Hamza, pp. 448 - 452.

[6] J.H. ter Bekke and J.A. Bakker, Fast Recursive Data Processing in Graphs Using Reduction, *Proceedings International Conference on Applied Informatics* (AI 2003), Innsbruck, Austria (2003).

[7] J.H. ter Bekke and J.A. Bakker, Modeling and Querying Recursive Data Structures I: Introduction, *Proceedings International Conference on Artificial Intelligence and Soft Computing* (ASC 2003), Banff, Canada (2003).