ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Jessie Kennedy and Peter Barclay (Eds)

Interfaces to Databases (IDS-3)

Proceedings of the 3rd International Workshop on Interfaces to
Databases, Napier University, Edinburgh, 8-10 July 1996

Paper:

The Use of the Data Dictionary in DBMS Based on Graphs

Marc Gemis

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

The Use of the Data Dictionary in DBMS based on Graphs

Marc Gemis

Dept. of Mathematics and Computer Science

University of Antwerp (UIA)

Universiteitsplein 1

2610 Wilrijk-Antwerpen, Belgium

Email: Marc.Gemis@uia.ua.ac.be

July 29, 1996

Abstract

The user of a DBMS has a collection of schemas, programs, methods, etc. . . stored in different locations, all over her computer. At the moment no DBMS offers appropriate tools to manage these collections. In this paper, we propose a graph-based data dictionary for graph-based database models and languages. This data dictionary can be used as basis for tools that assists the user in searching large amounts of meta-information. Some example tools are proposed. Two types of aggregation are investigated as mechanisms to make the query graphs more compact. The first is based on the aggregation found in the entity-relationship model, the second uses the fact that the data-dictionary describes graphs.

1 Introduction

Graph based representations of database schemas gain more and more popularity ([4] for an overview). In such a model classes are usually represented by labeled nodes, while properties and relationships are indicated by directed, labeled edges between the nodes. Experiments [6] show that these models allow the user to work faster and with less faults.

Most of those languages can only be used to formulate queries (e.g. SUPER [3], ICONICBROWSER [15], [12], . . .). GOOD [11] and PaMal [9] are graph-based languages that also allow to specify updates.

Most commercial database systems are based on the relational model. While the creation of the schema can be done by specifying an entity-relationship diagram; querying and manipulation is done through a textual 4GL language or SQL-statements. Those programs are stored in files, outside the database system. The only access method from within the database environment is through a directory/file browser which displays the names of the files. There are no search facilities provided.

The search facilities offered by the operating system such as WAIS, Harvest[5] with essence [13] and glimpse [14, 16], cannot be accessed from within the database system, and do therefore not offer satisfiable solutions.

It is well-known that the data dictionary is a database that contains information on user-defined schema. This information can be queried in the same way as user-defined schema. So far, programs are not stored in the data dictionary. Probably one of the reasons is that the structure of a (SQL-) program is completely different from that of a schema, and maybe the structure does not fit well into tables.

In this paper we will investigate how we can use the data dictionary to store information on graph-based programs. We show first that it is easy to define a data dictionary to store schema graphs. This model can easily be extended to store sequences of graphs, so that programs can be stored in the DBMS.

This allows us to write programs that manipulate programs stored in the data dictionary, it even allows us to write self-modifying programs.

The program graphs become rather large and for that reason we investigate techniques to shrink the graphs. The first technique is based on the idea of aggregation found in e.g. the entity-relationship model. The second technique

uses the fact that part of the data dictionary is used to describe graphs on a different level. By eliminating this level and using the graphs, we obtain a more compact representation.

2 A Graph-based Model

In this section we will give an informal description of a graph-based model for database modeling. It is a simple model, without many features. However, it is sufficient for its purpose in this paper. The presented model is based on PaMaL [9], which describes a graph-based language in a formal way.

2.1 The Schema

Figure 1: A schema for information on dogs.

It is known that the schema of an object-based database can be modeled using a directed labeled graph. The classes are represented as labeled nodes, while the properties are modeled as directed labeled edges between those nodes. In Figure 1, we show an example of such graph-based representation of the schema of a small database. It contains information on dogs. The schema contains the classes *Dog*, *Litter*, *Breed*, etc ... In the formalism we use here, these classes are represented in rectangles. The system defined classes are depicted in rectangles with rounded corners. Examples are *String* and *Date*. The type of a class *X* is depicted by a special shaped node. It can be a $\oplus$ or $\otimes$ -shaped node. The former indicates that the type is a tuple, while the latter indicates a set type. The node representing the type of the class and the node labeled by the class name are connected by a *typ*-edge. The attributes of a tuple node are given by the leaving edges. The type of the attribute is given by the end node of the associated edge. For example the class *Breed* has a tuple type with two attributes: *name* and *country*. The type of the elements in a set is given by the end node of the $\in$ -edge leaving the set node. An example is the *children*-attribute of the type of the *Litter* class. This attribute is a set of *Dog* objects. Furthermore, we can represent the class hierarchy with special edges. In this paper we use “outlined” edges, i. e. edges consisting of two lines. We label those edges with *isa*. In

the schema of Figure 1 we see that the classes `Dam` and `Sire` are subclasses of the class `Dog`. This means that the former two classes will inherit all attributes of the class `Dog`.

For simplicity, we demand that edges labeled by `typ` and `isa` and $\in$ are not used for other purposes than the ones stated above.

2.2 The Instance

Figure 2: An instance with three dogs.

Conceptually, we can also represent the instance as a graph. The composed objects (the elements of the classes) can be represented as nodes (*object nodes*). In the model we present here, we depict them as rectangles, labeled by the name of the class and the object identifier (OID). Examples of objects in the very small instance shown in Figure 2, are `Litter/L123`, `Sire/S23`. The atomic values, i.e. the objects belonging to the system defined classes, are represented as rectangles with rounded corner. An example is the value `String/Junior`. The graph-based representation of a tuple object is composed of the tuple node ($\oplus$), the leaving edges and the end nodes, which represent the values of the attributes. The sets objects are represented by the graph composed of the $\otimes$ -node, the leaving $\in$ -edges and the end nodes. The latter describe the values in the set. The value of a composed object is depicted by the node where the `val`-edge leaving the object node, arrives.

3 The Graph-based Manipulation Language

While the database designer can use the graph-based models (e.g. based on the entity-relationship model) of some commercial products to design her database scheme, querying the database has to be done with a QBE-like [17] interface or with SQL or a 4GL language. In the past we have presented graph-based alternatives [9, 11] for those traditional query languages. We will informally present a subset of PaMaL here.

A database operation has to specify which objects or tuple or records or ... have to be retrieved and eventually, how they have to be modified. This is the case for all operations, independent of the query language. To select the objects we use a *pattern-matching* technique. A *pattern* is a directed labeled graph. The types of nodes and edges that can appear in a pattern, are those that are used to described schemas and instances. The objects of the instance that are selected by a pattern, are those objects that belong to a subgraph of the instance, where the complete pattern is embedded. An example is shown in Figure 3.

Figure 3: Pattern to find the names all dogs that are mother of a litter.

The operations on the selected objects are described by nodes and edges that are drawn around the pattern. In order to distinguish them from the pattern nodes and edges, they will be drawn in a different shape. We use bold lines to indicate nodes and edges that have to be added, while nodes and edges in a dotted shape indicate deletion.

The addition of nodes means the creation of objects. Similarly the creation of edges means property assignment, node deletion is object deletion and edge deletion has to be interpreted as the removal of a property.

An example addition is shown in Figure 4. The pattern (the non-bold part), fixes all Breeds, whose country of origin is the U.K. The bold part indicates that for each such breed, a new property *recognized* is created. This property is a set and will contain one element with value KCGB. So, the addition in Figure 4 has to be interpreted as “Indicate by a a *recognized*-property that all English breeds are recognized by the KCGB.

Figure 4: An addition that creates a new set property.

Those operations can be placed in a sequence, thereby forming a *program*. The result of a program is obtained by executing the operations one after another, in the given sequence. A procedure is a named sequence of operations with a parameter mechanism, which is also represented in as a graph. To call a procedure from within a program or a procedure we added a procedure call operation to the language. This operation is also completely depicted as a graph.

Note that the complete language can express a wide range of queries [11]. However, we have to admit that the language is not useful to express mathematical formulae. The language can be extended with textual elements [8, 1] for this end, although conceptually those textual elements can still be modeled in our graph-model.

4 Data Dictionary

4.1 Modeling The Schema

It is well known that the information of the schema can be represented in the model itself. Commercial relational DBMS store schema definition — the description of a set of tables — in tables. This can be done with our graph-based model as well. Part of this is shown in Figure 5. We can also model the graph describing the schema in a schema graph. This is shown in Figure 6. Although the latter can be helpful to manipulate the layout of the schema graph with programs written in the proposed graph-based manipulation language, the amount of information that has to be kept additionally limits the usefulness to do this in general purpose DBMS. An example of a query that can be posed with a data dictionary describing the graph of a schema is “Give all edges that start in a node with an X-coordinate bigger than 10.” These queries depend on the visual layout of the schema and might therefore be useful for users that rely on their photographic memory. Unfortunately we have not done experiments to test this in practice.

Figure 5: Data dictionary for classes and types.

Figure 6: Data dictionary on node/edge level for schema graphs.

4.2 Modeling The Instance

The instance can also be modeled in the data dictionary. The classes in the data dictionary are shown in Figure 7. Those classes allow one to manipulate the concepts such as *object*, *tuple value*, ... The main drawback is that even for very small instances, the data dictionary representation is huge (See Figure 8). On the bright side, we do not need to store additional information, as the mapping of the data dictionary representation to the (traditionally) stored information of the instance can be done by a program. This is not the case if we want to store information on the graph of the instance. will not do this, as it is not useful, since the user will never see the instance graph, and therefore she will not know how it looks exactly in terms of coordinates.

Figure 7: Data dictionary classes to represent instances.

(a) The original instance.

(b) The instance in the data dictionary.

Figure 8: Representing an instance in the data dictionary.

4.3 Modeling The Data Manipulation Language

While the previous sections described concepts that are either known or that does not seem useful, we think that describing the operations in the data dictionary is new and useful. With the appropriate tools one can maintain a collection of program concepts (programs, individual operations, procedures, views, ...) in the same way one maintains the instance. Traditional DBMS store the programs (the SQL queries) in a number of directories that in the best case can be searched for the filename. For some operating systems, more intelligent index/search facilities exist (see Section 1), but these are not incorporated in the DBMS. By storing the program concepts in the database, we have disposal over a collection of powerful retrieval tools. Again we can store the information on two levels. The first level will describe concepts such as program, program step, addition, ... The second level will describe the operation graphs. The former level will not have serious drawbacks on the storage space, as we already have to store the programs as graphs in one way or another. Storing the program concepts in the database will allow one to retrieve e.g. all programs that contain in the second step an addition of a node with label A (see Figure 9 where we use a straightforward model of a data dictionary containing information over programs). Thus instead of using a traditional “directory plus file” navigator to browse through the collection of procedures in order to find that one particular procedure call, we can use a database query tool to search through our collection of procedures. The main drawback is the size of the graphs needed to describe the queries (see Section 6).

Figure 9: Pattern for programs with an A-node in the second step.

4.4 Security and Database Access

Some relational DBMS store the access privileges in a table as well. This is perfectly possible with the graph-based languages as well, if we provide the appropriate classes and attributes.

One needs at least a class of users and a class of possible privileges (such as read, write). Furthermore, one needs to add attributes to those objects for which one wants to define access rights. In the data dictionary schema one can model both class access rights and /or class property access rights. Furthermore, the access can be altered for individual objects (i.e. classes and attributes) or for all objects.

4.5 Manipulating The Data Dictionary

As far as we know is querying the only direct manipulation that commercial relational DBMS allow on the data dictionary. All other changes are done by special commands. E.g. to grant the user permission to a particular table, one can invoke an SQL-command. This is not possible by adding a tuple to some table. The data dictionary tables are treated differently, compared to “normal” database tables. In case of the table describing tables and columns this will guarantee the integrity between the tables and the information in the data dictionary.

For classes in the data dictionary containing data on programs, we can allow access through the same commands that are used to access to the (normal) database classes, on condition that the programs are only stored in the data dictionary. This means that the user can write programs in the same graph-based language as she uses to manipulate her database data. She does not need to learn any additional commands. Imagine that for some reason the name of a class has changed from Addr to Address. In a traditional environment, this means that each individual program has to be examined and eventually changed. This task can easily be accomplished with a program.

Again the main drawback is the size of the operations. We will propose two different solutions for the problem in Section 6.

4.6 Combining the different parts

By combining all the above described parts of the data dictionary, we can write operations which are not easy to accomplish with a conventional system. Take a look at the operation in Figure 10. This operation will add the set $\{3, 5\}$ to all set-values of objects belonging to a class created by the program XXX.

The “first line” in the pattern looks for all programs with name XXX. On the “second and third lines” the pattern requires that those programs also have a step whose operation is an Addition. The “fourth line” and the right part of the “fifth line” fixes the label of the node that is added in this addition. The left part of that “line” indicates that we need a class with the same name as that label. The “sixth line” looks for objects that have a set as value and that belongs to a class determined in “line five”. The bold part of the picture represents the set that is added.

Figure 10: Update of an instance, depending on contents of a program.

5 Tools For Using The Data Dictionary

5.1 Tools for Locating Programs

By storing the data dictionary as any other database, the DBMS tools can access the information in it as the data of any other database. The tools developed for working with a graph-based language for DBMS [10] c can be used without changes to query or update the data dictionary.

The purpose of those tools is to provide alternatives for the tools needed to write, maintain and execute SQL queries. They allow the user to write graph-based programs, execute them on a database and view the results of queries.

While those tools can be used without adaptation to query the data dictionary, they need to be modified in order to pass a selected program, procedure call, ... to another tool. The other tool can be e.g. a editor for graph-based programs.

5.2 Tools for Locating Classes

In larger schema graphs, it can be hard to locate a particular class in the schema browser. One of the reasons is that the complete schema might not fit on the screen and that scrolling is needed. We need a tool that allows us to ask queries on the data dictionary part that describes the current schema. The schema browser can then scroll to display the class we have selected in the result of the query.

5.3 Tools for Database Management Tasks

We have seen that the user security can be modeled in the the data dictionary. This means that the user can write programs in the manipulation language to update this information. No additional commands need to be learned.

6 The Graph Size Problem

We have seen that the graphs needed to work with information in the data dictionary tend to be large. In this section we discuss two techniques that deal with this size problem.

6.1 Aggregation

This technique hides structural information by grouping objects, attributes and relations into new objects. The interface can provide commands to switch between the visibility of the different levels.

Figure 11 shows the same schema viewed on two different aggregation or abstraction levels.

Any set of nodes and edges can be replaced by an *aggregation node*, which is drawn as a shadowed box. In Figure 11(b) the aggregation node A-Program¹ represents the Program-class and several of its attributes, as can be seen in Figure 11(a).

An *aggregation edge* represents all paths from one node to a collection of other nodes. Both the start node and the end nodes can be part of an aggregation node, in which case the aggregation edge starts or ends in the aggregation node. An example of an aggregation edge is the *steps*-edge in Figure 11(b). It is drawn as a double, waved line. This edge represents the path from the Program-node to the A-Step-node. Since the Program-node is part of the aggregation node A-Program, the edge starts in the latter node.

QBD* [2], which is a model and query language based on the ER-model, has a similar aggregation technique. The main benefit of our representation is that we use a different shape for aggregation nodes and edges. This mean that the user knows, without undertaking additional actions, that there is some hidden information.

¹We could have labeled this node with Program as well, but this would have been confusing during the explanation.

(a) A view on the schema with some detail.

(b) Viewing the schema at a very abstract level.

Figure 11: A database schema at different abstraction levels.

Aggregation can also be used in patterns and operations. This allows one to have more compact graphs for both patterns and operations. The compactification can be very strong, so that it can become difficult to understand which classes and properties participate in the aggregation, even with a good knowledge of the schema.

An example of an operation with aggregation is shown in Figure 12. It is the same operation as in Figure 10. Note that from this figure it is not obvious that the aggregation node `class`, represents the bold node of some program step. This is hidden in the definition of the aggregation edge `has`. Therefore, expansion to the complete pattern should always be possible.

6.2 Graph Aggregation

The attributes of a program are on one hand atomic (e.g. the name is a string) on the other hand there are composed attributes, such as the set of steps. Each program step has a name (atomic attribute) and an attribute describing the graph. The size of the data dictionary instance is mainly due to the part that describes the graphs of steps, patterns and the like. It is obvious that these graphs can be represented as a graph (namely by the graph self). This is shown in Figure 13. To distinguish the graph representation of the step from the other parts of the data dictionary instance, we put it in a box. We call such a box a *data dictionary graph-box* or *graph-box* for short.

The same technique can be used to represent the pattern, schema and instance graphs in the data dictionary instance. It is obvious that this concept can be used in a pattern on the data dictionary, as well as in programs on the data dictionary.

In order to be able to express a query like “Give all classes with an α -property”, we need to extend our graph formalism with nodes and edges without labels. When such a node (respectively edge) is used in the graph-box of a pattern, it will be embedded to all nodes (respectively edges) of the data dictionary instance graph. An example of such a pattern is shown in Figure 14.

To express an operation on the nodes or edges in a step, we will need another extension of the PaMaL formalism. In Figure 15 we add a node. This is done by putting a bold box around the node and edges that have to be added. This is the same operation as in Figure 16, where we use the plain data dictionary representation.

Similarly, a dashed box can be used to delete information from the data dictionary instance.

Figure 12: Update and instance using the data dictionary and aggregation.

Figure 13: Using a graph-box to express the query “Give all programs where in the second step the graph consisting of an A and a B-node, connected by a α -edge, is part of the pattern”.

Figure 14: All programs where the second step has a pattern with a node in which an α -edge starts.

Figure 15: Expressing the addition of a bold node and edge using a graph-box.

7 Related Work

Since those languages do not contain a manipulation language, they cannot be used for making adaptation to the stored programs. Reflection is not possible in those languages.

As we already mentioned, our aggregation technique is similar to the one used in QBD* [2] and the one found in the entity-relationship model. We are not aware of previous work using similar techniques as our graph aggregation. It is obvious that all graph-based languages that use patterns can be extended with a similar feature.

8 Implementation

9 Conclusions

The model and the language we used are both graph-based. This made the task of modeling the programs in the model a straightforward task.

The main drawback of the graph-based approach is the size of the program graphs. We proposed two solutions for this. A first approach is based on the classic aggregation technique, the second is based on the fact that the steps are graphs and can be represented as such.

Currently, we have an implementation of a prototype called xgood [8], which allows the user to formulate queries to retrieve a list programs that can then be loaded into an editor for further manipulation.

References

- [1] M. Andries. *Graph Rewrite Systems and Visual Database Languages*. PhD thesis, University of Leiden, The Netherlands, 1996.
- [2] M. Angelaccio, T. Catarci, and G. Santucci. QBD*: A fully visual query system. *Journal of Visual Languages and Computing*, 1(2):255–273, 1990.
- [3] A. Auddino, Y. Dennebouy, Y. Dupont, E. Fontana, S. Spaccapietra, and Z. Tari. SUPER: A comprehensive approach to database visual interfaces. In *IFIP 2nd Working Conference on Visual Database Systems*, pages 359–374, 1991.
- [4] C. Batini, T. Catarci, M.F. Costabile, and S. Levialdi. Visual query systems. Technical Report 04.91, University of Roma, "La Sapienza", Italy, 1991.
- [5] C.M. Bowman, P.B. Danzig, D.R. Hardy, U. Mamber, M.F. Schwartz, and D.P. Wessels. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, University of Colorado – Boulder, 1995.
- [6] T. Catarci and G. Santucci. Diagrammatic vs. textual query languages: A comparative experiment. In *IFIP 2.6: Third Working Conference on Visual Database Systems (VDB-3)*, pages 57–74, March 27–29 1995. (Preliminary Proceedings).
- [7] R. Cooper, editor. *Interfaces to Database Systems*, Workshops in Computing. Springer-Verlag, 1993.
- [8] M. Gemis. *Graph-based Languages in DBMS*. PhD thesis, University of Antwerp (UIA), 1996. Not published yet.
- [9] M. Gemis and J. Paredaens. An object-oriented pattern matching language. volume 742 of *Lecture Notes in Computer Science*, pages 339–355. Springer-Verlag, November 1993.
- [10] M. Gemis, J. Paredaens, and I. Thyssens. A visual database management interface based on GOOD. In Cooper [7], pages 155–175.
- [11] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A graph-oriented database model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, August 1994.
- [12] M. Halper, J. Geller, Y Perl, and E.J. Neuhold. A graphical schema representation for object-oriented databases. In Cooper [7], pages 282–307.
- [13] D.R. Hardy. Scalable internet resource discovery among diverse information. Master's thesis, University of Colorado – Boulder, May 1993.
- [14] S. Manber and S Wu. GLIMPSE: A tool to search through entire file systems. In *Usenix Winter 1994 Technical Conference*, pages 23 – 32.
- [15] K. Tsuda, M. Hirakawa, M. Tanaka, and T. Ichikawa. Iconic browser: An iconic retrieval system for object-oriented databases. *Journal of Visual Languages and Computing*, 1(1):59–76, 1990.

- [16] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, (35):83–91, October 1992.
- [17] M. Zloof. Query-by-example: a data base language. *IBM Systems Journal*, 16(4):324–343, 1977.