Implementation of a Graph-Based Data Model for Complex Objects

Mark Levene†, Alexandra Poulovassilis‡, Kerima Benkerimi†, Sara Schwartz†, Eran Tuv†

† Department of Computer Science, University College London,
Gower Street, London, WC1E 6BT, U.K.
E-mail:M.Levene@uk.ac.ucl.cs.

‡ Department of Computer Science, King's College London,
Strand, London, WC2R 2LS, U.K.
E-mail:alex@uk.ac.kcl.dcs

Abstract

We have developed a graph-based data model called the Hypernode Model whose single data structure is the *hypernode*, a directed graph whose nodes may themselves reference further directed graphs. A prototype database system supporting this model is being developed at London University as part of a project whose aims are threefold : (i) to ascertain the expressiveness and flexibility of the hypernode model, (ii) to experiment with various querying paradigms for this model, and (iii) to investigate the suitability of the directed graph as a data structure supported throughout all levels of the implementation. The purpose of this paper is to report upon our findings to date.

1. Introduction

Much recent database research has focussed on deductive and object-oriented databases and also on graph-based representation and manipulation of data. Following the general direction of these trends, we have developed a graph-based data model called the Hypernode Model which supports object identity and arbitrarily complex objects.

Other data models have also used graphs as their underlying data structure. However, a feature common to all these models is that the database consists of a single graph. This has the drawback that complex objects consisting of many inter-connected nodes are hard to represent in a clear way. In contrast, a hypernode database is a set of *interconnected* graphs. This unique feature of the model provides inherent support for data abstraction and allows each real-world object to be represented as a separate database graph, both of which are significant aids to data representation and manipulation.

The hypernode data structure was first described in [2] while in [7] we described a graph-based language that can be used for querying and updating hypernode databases. This work was consolidated in [3]. In [8] the model and language were extended with types and an analysis of the efficiency of type checking and of inference was carried out. In [6] a procedural language for the model was defined and an account of functional dependencies for the hypernode model was given. In [6, 8] the hypernode model was proposed as a suitable underlying formalism for hypertext. A detailed comparison of the hypernode model with other graph-based models can be found in [8]. Finally in [1,9] the implementation of a prototype database system based on the hypernode model is described.

The structure of this paper is as follows. In Section 2.1 we present the hypernode data structure, hypernode databases, and schemas for these. In Section 2.2 we present and illustrate two query languages for the model, a procedural language called HNQL and a rule-based language called Hyperlog. In Section 3 we describe the implementation of our prototype database system, addressing its physical, conceptual and external levels in turn. Finally, in Section 4 we discuss our contribution and list several directions in which research is proceeding.

2. The Hypernode Model

2.1. Hypernode Databases and Hypernode Schemas

For the purposes of defining the hypernode model, we assume the availability of two disjoint countable sets of constants, a set **L** of labels and a set **P** of primitive nodes. We denote elements of the set **L** by identifiers which start with an upper case letter and elements of the set **P** by identifiers which start with a $ or a lower case letter or by strings delimited by double quotes.

A *hypernode* is an equation of the form

$$G = (N, E)$$

where $G \in \mathbf{L}$ is termed the *defining label* (or simply the label) of the hypernode and (N, E) is a digraph such that $N \subseteq (\mathbf{P} \cup \mathbf{L})$. (N,E) is termed *the digraph of* *the hypernode* (or simply the digraph corresponding to G)

A *hypernode database* (or simply a database), HD, is a finite set of hypernodes that satisfies the following two conditions:

(H1) no two distinct hypernodes in HD have the same label;

(H2) for any label G in the node set of a digraph of some hypernode in HD there exists a hypernode in HD with label G.

Condition H1 corresponds to Codd's entity integrity since each hypernode can viewed as representing a real-world entity. In object-oriented database terminology, labels are unique and serve as object identifiers. Similarly, condition H2 corresponds to Codd's referential integrity, since it requires that only existing entities be referenced.

We illustrate the above definitions in the Appendix where we show part of a passengers and airlines database. In particular the three windows in the left-hand column show :

(1) A hypernode that represents an entity set, namely a number of PASSENGER entities.

(2) A hypernode that represents a binary relationship between passengers and the airlines they are currently booked to fly with.

(3) A hypernode encapsulating a passenger record with the relevant attribute-value pairs. We node that these hypernodes are bipartite digraphs whose node sets are partitioned into attribute names and attribute values (we adopt the convention of primitive nodes starting with a $ for attribute names).

We observe that hypernodes can contain the labels of other hypernodes, and can thus be conceptualised as *nested* digraphs (this is the view that we adopted and expounded upon in [3, 8]).

Also, if we view hypernodes as representing either entity sets, entities, or binary relationships, then the hypernode model can be regarded as a formalisation of the instance level of the Entity-Relationship model, extended to allow nesting of entities and relationships.

Analogously to hypernodes and databases above, we define *hypernode types* (or simply types) and *hypernode database schemas* (or simply schemas), respectively. For this, we assume the availability of two disjoint countable sets of constants : a set of type labels **TL** whose instances are labels from **L**, and a set of primitive types **TP** whose instances are primitive nodes from **P**. A *type* is then an equation of the form

$$T = (M, F)$$

where $T \in$ **TL** is the *label* of the type and (M, F) is a digraph such that $M \subseteq (\mathbf{TP} \cup \mathbf{TL})$.

We say that a hypernode G = (N, E) *is of type* T = (M, F) if there exists a total and onto function $\theta: N \rightarrow M$ that preserves both adjacency and also the types of labels and primitive nodes. We say that a database HD *is over a schema* HS if $\forall$ G = (N, E) $\in$ HD $\exists$ T = (M, F) $\in$ HS such that G = (N, E) is of type T = (M, F).

We illustrate types in the Appendix where we show part of the schema for the passengers and airlines database. In particular, the three windows in the right-hand column show :

(1) The type SET_OF_PASSENGERS.

(2) The type FLIES_REL.

(3) The type PASSENGER_REC.

It is easily verified that each hypernode illustrated in the left-hand column is of the type shown next to it.

In [8] we discussed the expressiveness of types. In particular, as we have illustrated above, set types, binary relationship types, and nested record types are all representable. Also, the hypernode is *type-complete* in the sense that the only allowed type-forming operator (digraph definition) can be applied arbitrarily many times. Furthermore, schemas are also digraphs and thus the meta data can be queried and updated using the same formalism as the data.

2.2. Query Languages for the Hypernode Model

We have defined two query (and update) languages for the hypernode model : a rule-based query language called Hyperlog and a procedural language called HNQL.

Hyperlog programs consist of a finite set of rules. The body of each rule consists of a number of digraphs, called *templates*, which may contain variables and are matched against the hypernodes in the database. The head of each rule is also a template and indicates the updates (if any) to be undertaken for each match of the templates in the body. The evaluation of a program comprises a repeated matching of its rules against the database (in parallel) until no more updates can be inferred and a fixpoint is attained. A comprehensive description of the syntax and semantics of Hyperlog, and a comparison with related database languages can be found in [8].

HNQL consists of a basic set of primitive operators for updating hypernodes and for testing the membership of nodes and arcs in hypernodes. In addition to these deterministic operators, several non-deterministic operators are provided for the arbitrary selection of a node or an arc from a set thereof. HNQL is further extended into a procedural query language by adding assignment, sequential composition, a conditional construct, and bounded and unbounded iteration. A comprehensive description of the syntax and semantics of HNQL can be found in [6].

For the purposes of both these query languages we assume the availability of a countable set of variables, **V**, and denote variables by uppercase identifiers from the end of the alphabet. We introduce the flavour of the languages via two simple examples.

The first example is an HNQL program, shown in Figure 1, that selects the names of passengers who are flying on flight number "BA212" and puts them into a new hypernode whose label is RESULT. The counterpart Hyperlog program is shown in Figure 2.

```
X1 := create();
X2 := rename(X1, RESULT);
X2 := insert_node(RESULT, $name);
for_all X1 ∈ nodes(PASSENGERS) do
  if ($flight_no, "BA212") ∈ arcs(X1) then
    for_all (Y1, Y2) ∈ arcs(X1) do
    TB
      if Y1 = $name then
      TB
        X2 := insert_node(RESULT, Y2);
        X2 := insert_arc(RESULT, $name, Y2);
      TE
    TE
```

Figure 1.

```
RESULT = ({$name, Y}, {($name, Y)}) ←
  X = ({$flight, "BA212",$name,Y},
      {($flight, "BA212"),($name,Y)}),
  PASSENGERS = ({X}, ∅)
```

Figure 2.

The second example is an HNQL program, shown in Figure 3, that modifies flight number "BA212" to "BA345" for all passengers in the database. The counterpart Hyperlog program is shown in Figure 4.

```
for_all X1 ∈ nodes(PASSENGERS) do
  for_all (Y1, Y2) ∈ arcs(X1) do
  TB
    if (Y1, Y2) = ($flight_no, "BA212") then
    TB
      X2 := delete_arc(X1, Y1, Y2);
      X2 := delete_node(X1, Y2);
      X2 := insert_node(X1, "BA345");
      X2 := insert_arc(X1, Y1, "BA345");
    TE
  TE
```

Figure 3.

```
X = ({¬"BA212","BA345"},{($flight,"BA345")})←
  X = ({$flight,"BA212"}, {($flight,"BA212")}),
  PASSENGERS = ({X},∅)
```

Figure 4.

Not surprisingly, the Hyperlog programs are more compact than their HNQL equivalents. They are also more suitable for end users since they can easily be represented graphically. Conversely, sequencing is expressed more easily in HNQL which is consequently more useful for low-level computations (although see [8] for an example of how to simulate counter programs in Hyperlog). We have also shown in [8] and [6] respectively that both Hyperlog and HNQL are *update-complete* i.e. all computable updates can be expressed in both languages.

3. Implementation

In this section we describe the implementation of a prototype database system based on the hypernode model. The main aim of this prototype has been to investigate the suitability of the directed graph as a data structure supported throughout all levels of the implementation. We address each of the three levels of the implementation in turn in the following three sub-sections.

3.1. The Physical Level

The physical level is a hypernode database consisting of two components, a *storage manager* and an *index manager*. The main effort at this level has been directed towards developing a modular and extensible storage manager which can be used as a reliable and stable core for future versions of the software. In particular, the storage manager caters for :

- object-identity and referential sharing between hypernodes;
- efficient update of hypernodes, including *large* hypernodes, i.e. ones with a large node set or with one or more large primitive nodes, and *complex* hypernodes, i.e. ones with a dense edge set;
- clustering strategies for hypernodes in secondary storage;
- buffering of hypernodes in main memory;
- retrieval operations which utilise indexing provided by the index manager.

The main novelty of the storage manager compared with other persistent stores is its use and optimisation of a single graph data structure. The software is a library of C++ classes which provides four categories of operations for : creating, updating, traversing, and associative querying of a hypernode database. A comprehensive description of the implementation of the storage manager can be found in [9].

3.2. The Conceptual Level

The conceptual level is the Hyperlog interpreter. The main effort here has been directed towards developing efficient algorithms for the matching of templates in rules against the database. A bottle neck that arises in this matching process is the calculation of permutations of node sets - in [8] we discuss several heuristics which can be used to ameliorate this problem. We also discuss there how type checking of Hyperlog programs can be performed statically and in polynomial time.

The software for the Hyperlog interpreter is a library of C++ classes which provides three categories of operations for : matching templates in rule bodies, performing the updates arising from rule heads, and composing these for the overall processing of rules and programs.

Recently, we have developed a semi-naive evaluation algorithm for Hyperlog programs without deletions which is analogous to the well-known algorithm for the (much simpler) Datalog language. In the absence of EDB and IDB predicates, this is achieved by syntactically partitioning the templates in the bodies of Hyperlog rules into those templates which cannot be affected by any updates to the database during the evaluation of a program and those templates which may be affected by updates to the database. The former templates play a role similar to that of EDB predicates and the latter similar to that of IDB predicates in relational databases. A description of this work can be found in [1].

3.3. The External Level

A graphical interface for Hyperlog is currently being developed at this level. Also, a hypertext interface which we call HyperGUI has been implemented using Motif and X-windows. HyperGUI allows the user to build up a database of hypernodes and to update and navigate through the hypernodes of an existing database. Updates are performed via menus that allow the user to add and delete nodes and arcs in the current hypernode. Navigation is performed by clicking on buttons that represent the labels of hypernodes, causing a window containing the nodes and arcs of the selected hypernode to be displayed. Two additional windows are displayed in order to help the user navigate through the database. The first of these is an overview diagram which contains all the labels of hypernodes that are directly referencing or being referenced by the current hypernode being browsed. The second window lists all the labels of hypernodes in the database, allowing simple random access to any hypernode. Browsing of a textual fragment is performed by displaying the text file containing the text in a separate window. In the appendix we show several screen dumps of hypernodes that have been created using HyperGUI.

4. Conclusions

We have shown the viability of developing a database system for the hypernode model which is based on graphs throughout all its levels, from the user interface down to the physical level :

- At the physical level we have the developed a repository (the storage manager) which is well suited to the storage and update of hypernodes.
- At the conceptual level we have developed a query and update language (Hyperlog) which supports both queries and updates.
- At the external level we have developed a graphical interface (HyperGUI) which provides a user-friendly way to set up, update, and navigate a database of hypernodes. We are also implementing a graphical interface for Hyperlog.

The results of the hypernode project contribute towards general research into graph-based database systems. We particularly envisage the suitability of the hypernode model for applications such as CASE, hypertext, and knowledge bases. With respect to CASE, we are developing an integrated data and process model for modelling the software development process [5]. There are also several open problems that have arisen from the project and require further investigation :

- We would like to extend the type system of [8] to cater for structural inheritance. This is an important feature that can naturally be supported by our nested graph structures (as discussed in [4] for a predecessor of the Hypernode Model).
- An ongoing research problem is that of query optimisation for query languages for complex objects. In particular, the work of [1] needs to be extended to cater for deletions as well as insertions. Also, at the physical level the index manager needs to be extended with more sophisticated indexing suitable for the graph matching that underlies Hyperlog.
- We would also like to investigate other kinds of query paradigms for the Hypernode Model e.g. functional and object-oriented ones ([2] gives a preliminary discussion of these).
- Finally, we feel that the results of this project provide a sound foundation for further work in the emerging research into hypertext, in particular into integrating navigational and declarative queries over hypertext databases.

Acknowledgements

This work has been funded by Grant GR/G26662 of the U.K. Science and Engineering Research Council. We would like to thank our colleagues George Loizou and Ray Offen for many fruitful discussions.

References

[1] Benkerimi K. and Poulovassilis A. 1993. Semi-Naive Evaluation for Hyperlog, a graph-based language for complex objects. To appear in *Proc. 1st International Workshop on Rules in Database Systems*, Ediburgh. Springer-Verlag.

[2] Levene M. and Poulovassilis A. 1989. The hypernode model - A graphical approach to integrating data and computation. In *Proc. 1st International Workshop on Foundations of Models and Languages for Data and Objects*, Aigen, Austria, pp. 55-77.

[3] Levene M. and Poulovassilis A. 1990. The hypernode model and its associated query language. *Proc. Fifth Jerusalem Conference on Information Technology*, pp 520-530.

[4] Levene M. and Poulovassilis A. 1991. An object-oriented data model formalised through hypergraphs. *Data & Knowledge Engineering*, Vol. 6, No. 3, pp. 205-224.

[5] Levene M. and Offen R. 1992. A unified graph-based data and process model. Research Note RN/92/99, Dept. Computer Science, University College London.

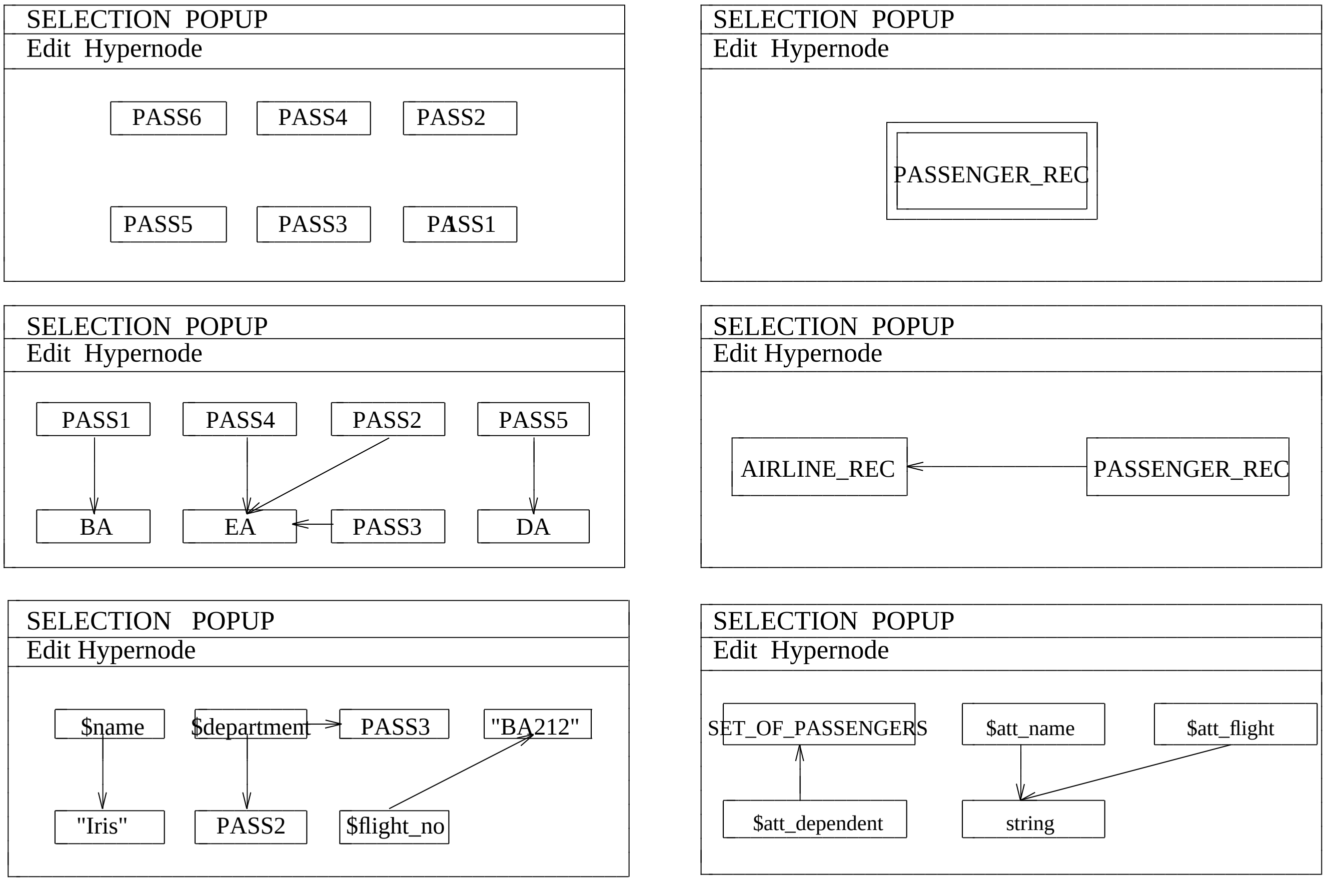
[6] Levene M. and Loizou G. 1993. A graph-based data model and its ramifications. *IEEE Transactions on Knowledge and Data Engineering*, in press.

[7] Poulovassilis A. and Levene M. 1989. Hypernode programs as the basis for a graphical query language. Research Note RN/89/87, Dept. Computer Science, University College London.

[8] Poulovassilis A. and Levene M. 1993. A nested-graph model for the representation and manipulation of complex objects. *ACM Transactions on Information Systems*, in press.

[9] Tuv E., Poulovassilis A. and Levene M. 1992. A storage manager for the hypernode model. In *Proc. 10th British National Conference on Databases*, Aberdeen, pp. 59-77. Springer-Verlag LNCS Vol. 618.

Appendix


SELECTION POPUP
Edit Hypernode
PASS6
PASS4
PASS2
PASS5
PASS3
PASS1
SELECTION POPUP
Edit Hypernode
PASSENGER_REC
SELECTION POPUP
Edit Hypernode
PASS1
PASS4
PASS2
PASS5
BA
EA
PASS3
DA
SELECTION POPUP
Edit Hypernode
AIRLINE_REC
PASSENGER_REC
SELECTION POPUP
Edit Hypernode
$name
$department
PASS3
"BA212"
"Iris"
PASS2
$flight_no
SELECTION POPUP
Edit Hypernode
SET_OF_PASSENGERS
$att_name
$att_flight
$att_dependent
string