A Schema-Level Representation for Functional Databases Using Lattices

J L Campos dos Santos and P J H King
Birkbeck College, University of London
Department of Computer Science
Malet Street, London WC1E 7HX, England
email: [ubacr28, ubacw02]@dcs.bbk.ac.uk

Abstract

This paper presents an approach to providing an interface to a functional database which allows novice users to query the database without the need to be familiar with a query language or the database schema. We use the set of types and functions in the database schema as the basis for generating a set of default queries. This set of default queries may then be modified by the database designer to select the most appropriate queries and to add further queries which were not automatically generated. The queries are organised in the form of a lattice such that similar queries are closely related. This lattice then forms the basis of a graphical interface to the database from which queries can be invoked and further refined.
1 Introduction

Retrieving information from a database is hard for novice users for two reasons. Firstly they must learn some query language and most such languages are opaque to the average user. Secondly the user must be familiar with the way in which information is modelled in the schema. Since a database schema reflects the collective needs of various sets of users it is likely that a given schema will not correspond to an individual user’s perception of the data. However, it is possible to use a database schema as the basis for generating a set of default queries. This is particularly so for a functional database where the schema organisation closely models the semantics of an application domain.

In this paper we present a way of generating database queries from a functional database schema. These queries may then form a starting point for a user to explore the database. The strategy we adopt has four stages:

- A functional database schema is transformed into a set of subschemas which are organised in the form of a lattice. These subschemas have default queries associated with them.

- The set of subschemas and default queries may be modified by the database designer in order to remove queries which are not useful or to add further queries.

- The lattice of queries then forms the basis for a user interface to the database. The user may explore the lattice and execute the query associated with a node in order to ‘descend’ to the instance-level data.

- Further facilities are provided that allow the user to refine the query at the instance-level.

Our system supports multiple lattice-views of the same database. Hence this approach allows a database designer to tailor different sets of default queries (organised in lattice-form) appropriate to the needs of different groups of users.
In the rest of this paper we first introduce our functional data model and a running example based on a database of election results. We then describe how we generate a set of subschemas, how these subschemas are organised in a lattice and the default queries associated with lattice nodes. We then briefly review the facilities which allow the interface designer to tailor the lattice. Finally we discuss the advantages of our approach and outline areas for further research.

2 Our Data Model

We use a functional data model by which we mean that we take a functional view of a binary relational model of information. A binary relation \( R \) over two sets \( A \) and \( B \) can be viewed in terms of two functions, say \( f : A \to 2^B \) and \( g : B \to 2^A \), such that if the pair \((a, b) \in R\) then \( b \in f(a) \) and \( a \in g(b) \). Some relations (those which intrinsically correspond to functions) can be represented by single-valued functions of the form \( h : A \to B \) rather than set-valued ones. In our model we view the information from a functional perspective in which relations are modelled by single-valued functions where possible and set-valued ones where not.

In our model only atomic types are supported — these are nonlexical types and lexical types. Nonlexical types are sets of object identifiers which serve as surrogates for entities of interest in the application domain. Nonlexical values have no meaningful printed representation. Lexical types, correspond to sets of values such as integers, or strings which do have a printable form. Our division of types into lexical and nonlexical follows that of NIAM [VV82].

Our model differs from previous functional models, such as those of [Shi81] and [BFN82] in that there is no support for constructed types. The reason for this is that we believe that the use of constructed types may obscure the underlying semantic relationships between atomic values. The standard objection to this approach, that it entails the introduction
of 'artificial' entities, does not concern us since such entities can be completely hidden in the views which we present to the user. On the other hand, the use of a strictly graph-oriented representation of data makes the schema more amenable to update and gives greater flexibility in defining user-views of the data. Moreover, constraining the representation of database information in this way does not compromise our ability to present information to the user in the most appropriate form. Once data is retrieved it can be organised, manipulated, and presented in the form of constructed values if required.

We take as example a simplified schema diagram of an Election¹ database which is shown in Figure 1. There are three non-lexical types: cr (Candidate Result), oc (Occupant of Seat), and pn (Person). The lexical types are: fromdate, tilldate, pname, and votes. The functions are: cd (Candidate), ip (Is Person), np (Name of Person), sf (Serving From), st (Served Till), vt (Votes Obtained). The functions defined between non-lexical entities are invertible, and those which return a set-valued result are known as multi-valued.

The application declarations in this paper use the syntax of the functional database language FDL [KP88, PK90, Pou92]. Type and function declarations constitute the schema of a functional database. The declarations of lexical types generate synonyms for the basic types. Types and functions are distinguished by their names, and hence these must be unique identifiers. The following declarations implement the application schema shown in Figure 1.

```plaintext
/* === Types declarations - Synonyms === */
fromdate  == integer;
tilldate == integer;
pname    == string;
votes    == integer;

/* === Non-lexical entities declarations === */
cr : : nonlex;
oc : : nonlex;
pn : : nonlex;
```

¹Our example is based on a real application of a database for recording election results in Greenwich Council, South London.
3 Representing the Database Schema in Lattice Form

In this section we introduce a lattice approach to representing the schema of a functional database. The method uses the set of Function names (F), the set of Nonlexical types (N), and the set of Lexical types (L) in the schema. For our running example the values of these three sets is as follows:

\[
\begin{align*}
N &= \{\text{cr, oc, pn}\} \\
F &= \{\text{cd, ip, np, sf, st, vt}\} \\
L &= \{\text{fromdate, tilldate, pname, votes}\}
\end{align*}
\]

We refer to the union of these three sets as the *NFL set*. Subsets of the NFL set correspond to subschemas. The total set of subschemas which may be generated by taking
subsets of NFL is the set $2^{NFL}$, though of course some of these sets will not correspond to valid or meaningful subschemas. We filter out the invalid subschemas — that is those which include functions whose domain or range type is missing. Given two subschemas, represented by the sets $S_1$ and $S_2$, then if $S_1 \subseteq S_2$ this is equivalent to saying that $S_1$ is a subschema of $S_2$.

The powerset of a set and the subset relation together form a lattice — a partially ordered set such that all pairs of elements in the set have a greatest lower bound and a least upper bound. Our lattice may be represented diagrammatically by a graph in which nodes represent sets (or subschemas) and an edge between two nodes denotes that the lower node is a subset of the higher node as shown, for instance, in Figure 4. The formal definition of lattices and their properties can be found in Draskovicova [Dra92].

Our motivation for viewing the set of subschemas of the database as a lattice is that it provides a way of presenting the database to a new or naive user. Subschemes which are towards the top of the lattice are larger and contain more of the semantics of the database and relationships between entities. Subschemes lower down in the lattice represent smaller or minimal subschemas.

However, there is the problem that the set of all valid subschemas is prohibitively large. For a set with $n$ elements there are $2^n$ different subsets so the number of valid subschemas of a schema is exponential in the size of the schema. To make it possible to present the user with a lattice of subschemas and associated default queries it is essential to reduce the number of subschemas to a workable size, preferably to a size linear in the size of the schema.

Our approach is based on filtering the set $2^{NFL}$ of all possible subschemas, to produce a meaningful set of subschemas which can be used for generating potentially useful default queries. Essentially we take two kinds of subschema which we term complex subschemas.
and *primitive subschemas*. A complex subschema corresponds to a nonlexical type, such as $oc$ and all the functions in the schema which have that type as their domain and where the ranges is a nonlexical type, all the functions which have that type as their range etc.

The corresponding subschema is shown in Figure 2. Primitive subschemas comprise a single function, such as $vt$, along with its domain and range as shown in Figure 3.

These two kinds of subschema correspond to queries which give all the information on a particular class of nonlexical entities or which show all the values in a lexical type respectively. Queries can be automatically generated and associated with a particular subschema as explained in the next Section.
Using this approach the set of subschemas which is generated for the schema in Figure 1 is shown below in tabular form and its lattice organisation in Figure 4.

<table>
<thead>
<tr>
<th>Node Label</th>
<th>Subschema</th>
</tr>
</thead>
<tbody>
<tr>
<td>End-Enlargement</td>
<td>{cd, cr, fromdate, ip, np, oc, pn, pname, sf, st, tilldate, votes, vt}</td>
</tr>
<tr>
<td>oc</td>
<td>{oc, sf, st, ip, fromdate, tilldate, pn}</td>
</tr>
<tr>
<td>cr</td>
<td>{cr, vt, cd, votes, pn}</td>
</tr>
<tr>
<td>cd</td>
<td>{cd, cr, pn}</td>
</tr>
<tr>
<td>ip</td>
<td>{ip, oc, pn}</td>
</tr>
<tr>
<td>np</td>
<td>{np, pn, pname}</td>
</tr>
<tr>
<td>sf</td>
<td>{sf, oc, fromdate}</td>
</tr>
<tr>
<td>st</td>
<td>{st, oc, tilldate}</td>
</tr>
<tr>
<td>vt</td>
<td>{vt, cr, votes}</td>
</tr>
</tbody>
</table>

End-Refinement {}  

Figure 4: The Lattice diagram for the schema level

4 Generating Default Queries

The lattice diagram represents the schema level data space of the application. Nodes towards the "End-Enlargement" have enlarged subschemas, whereas nodes towards the "End-Refinement" have refined subschemas.
The user can select a node to be the active node; this represents the present position of the user in the lattice diagram. The nodes that are linked to the active node with an edge are called neighbour nodes. The ability to explore a neighbourhood constitutes an important feature in browsing systems. In our approach it allows users to visit nodes which share common subschemas. This helps browsing the lattice, as the user knows that moving from an active node to a neighbouring node will only result in a small difference in the subschema content. Change by small steps is very useful for query reformulation.

A primitive node is one which comprises a primitive subschema, for example, the primitive subschema "\{vt, cr, votes\}". The label of a primitive node is given by its function name vt, in this instance. The default query for a primitive node is to display the contents of the function range. Hence for the vt node the query will be "For each cr print the value of vt(cr)". This corresponds to the FDL query,

\[
[ \text{vt} \ x \ |\ x <- \text{All}_{\text{cr}} ];
\]

A complex node is one which comprises a complex subschema such as "\{oc, sf, st, ip, fromdate, tilldate, pn\}". The label of a complex node is given by the nonlexical type which appears in the subschema; oc in this instance. The default query is generated from the node's content. The form of this query is "Return all the information held an each entity in the entity set". For the oc node this query, in FDL is,

\[
[ \{\text{np} \ ip \ x, \text{sf} \ x, \text{st} \ x\} |\ x <- \text{All}_{\text{oc}} ];
\]

Note that the function ip x returns a nonlexical which would not be meaningful to the user and so np ip x is used which provides understandable information about that nonlexical.

Navigation in the lattice is not limited to moving along the links. The structure provides facilities to locate a specific subschema, to tailor nodes, to select queries to be evaluated and to access the query results by browsing the instance sets.
5 Facilities for Tailoring the Interface

The lattice which is automatically generated, as described above, may be tailored by the
designer of the database interface. This tailoring takes two forms:

- the designer may remove nodes or add further nodes to the lattice. These additional
  nodes correspond to subschemas which were not automatically generated,

- the contents of a node may be changed — this may involve adding documentary
  information for the user, modifying the default query, or adding further queries.

This tailoring process allows the designer to construct alternative interfaces to the same
database.

Deleting a node from the lattice is equivalent to removing a subschema and associated
query which will be of no interest to an end user. Adding a node allows more complex
subschemas to be included, along with associated queries. Note that the designer is not
restricted in the form of subschema that they add, and that the addition of subschemas
may result in the lattice having more than two levels. However many levels are added,
the lattice organisation of the subschemas provides a ready-made set of routes between
alternative queries which may be used as a guide by the user.

The tailoring which may be carried out on an individual node includes altering the
default query to make it more appropriate, or adding further queries, provided these
queries are consistent with the subschema represented by the node. The designer may
also add further information to a node, such as descriptions of the functions which appear
in the subschema.

The lattice forms the basis of the database interface presented to the user. Each lattice
node becomes, in effect, a hypertext button which when clicked displays all the information
and queries associated with that node. The user may invoke any of the queries associated
with the node. This has the effect of allowing the user to descend to the instance level. The results from the query may be further refined by the user as required.

6 Conclusion

In this paper we have presented an approach to automatically generating and refining a set of queries on a functional database. These queries are naturally organised in a lattice which forms the basis of an end-user database interface. Our approach has been used in the implementation of the COCOON database browsing system which is implemented on a UNIX Workstation using the script language Tcl/Tk, a Motif-like user interface design language [Ous94], with the functional database language FDL as the underlying database manager. Currently experimentation is under way with the system as the interface to several databases including a more complete version of the election example used in this paper.

We believe that our approach has the following advantages:

- It forms the basis of a user interface which allows end-users to browse and query large databases without the need to know either a query language or the particular organisation of the data;

- It can form the basis of a view system in which a database designer can tailor different end-user interfaces which are appropriate to the needs and perceptions of different groups of end-users;

- The ability to present users with views of the database and insulate them from the actual organisation of data avoids the problem which arises in strictly binary-relational databases that 'artificial' entities must be defined to accommodate information which might be more naturally held in n-ary form. This is still the case but our system ensures that all end-users are shielded from any view of the data
which they might find counter-intuitive. On the other hand, adopting a strictly
graph-oriented organisation means, we believe that the semantics of the data is
more accurately modelled and that modifications to the schema are more easily
accommodated.

Areas of further research include:

- Investigating ways to accommodate schema updates (in particular the removal of
elements) without losing those tailored queries which are not affected by the change;

- Investigating ways in which queries specified by different lattice nodes can be com-
bined to produce more complex queries corresponding to complex paths through
the data.

The approach which we have adopted to providing a database interface is related to
some which have been used in document retrieval systems, where lattices occur naturally
when documents are compared by their key terms. Examples of this approach are found
in the SMART system [Sal71], and Godin’s browsing tool prototype [GSG86].

Acknowledgments

We are grateful to our colleagues in the TriStarp Group, and particularly to Robert
Ayres and Peter Rodgers for numerous discussions and continuous refinement of the ideas
presented in this paper. This work has been partially supported by the National Institute
of Amazonian Research (INPA), The Brazilian National Research Council (CNPq) - Brazil
and The British Council.

References

[BFN82] Peter Buneman, Robert E Frankel, and Rishiyer Nikhil. An implementation
technique for database query languages. ACM Transactions on Database Sys-


