A multi-level relational system

by J. MYLOPOULOS, S. SCHUSTER and D. TSICHRTIZIS

University of Toronto
Toronto, Ontario

INTRODUCTION

A relation can be conceptually viewed as a table of data. The table’s heading defines the relation’s name, the column headings are the attribute names, and each row corresponds to an n-tuple of data values describing a single entity. The set of values which can be used in a column is called a domain. A relational data base is composed of a set of time varying relations inter-related through common domains.

The relational model of data has been proposed as a flexible user interface to a data base management system (DBMS). Data languages based on relations are less procedural and more powerful than their existing counterparts in currently available systems. However, they leave optimization and access path decisions to the system. For example, the DBMS must decide how to best use many common mechanisms like pointer arrays, data pools, inverted files, and semantic networks in a manner transparent to the user.

We describe in this paper the mechanisms used in the development of a prototype system called ZETA/TORUS. We present in a hierarchical fashion a set of facilities which ultimately lead to the implementation of the complete system. Most of the techniques used are present in other systems. The important consideration is to isolate and outline their role in implementing relations.

This system is composed of three principal levels. The low level system implements elementary relational operations, inverted files, and the ability to “mark” subsets of relations. The intermediate level implements derived relations. It also has the ability to combine elementary queries of the lower system into high level relation operations. Finally the high level uses a compiler-compiler for query language generation, a preprocessor/compiler for a host language system, and a semantic network for a natural language understanding system.

THE PRIMITIVE RELATIONAL LEVEL

At this level the system provides a basic facility to manipulate relations. An “actual” relation can be created and is stored in a basic direct access file. Each tuple of the relation corresponds to a record of the file. Names of the relations and their domains and other characteristics are stored in system tables. Each one of the created relations has independent existence and it can be queried or updated through commands. All commands are procedure calls to this level of system. The format of the commands is very strict and users must abide by many conventions. The main purpose of this level is to implement some important mechanisms and not to provide a user interface.

The system provides a complete set of commands to deal with tuples within named relations. Namely the user has commands to:

(a) Create and destroy relations,
(b) Lock and unlock relations,
(c) Insert and delete tuples,
(d) Get and update data elements within a single tuple,
(e) Mark relations.

There are no commands applying to more than one relation at this level.

The system also provides a primitive data definition language. In addition, users are able to query, but not update, the schema tables and get the proper names for domains, relations, etc. This facility provides an elementary form of data dictionary.

The most interesting command of this level is the marking operation. A “mark” corresponds to a unary relation which stores indices of tuples of an actual relation which satisfy a Boolean qualification. We use this facility as a tool for the construction of more complex operations.

The unary relation is simply an array of indices related in two ways:

(a) The indices point to tuples of the same “actual” relation,
(b) The tuples referenced satisfy the same qualification.

These unary relations are implemented using a header and a body. The header contains control information while the body contains the indices.

Several basic operations are implemented to manipulate these structures. The user may create or destroy marks. He can retrieve tuples of the marked relation. At this point, modification of marks by the user is not allowed. Future extensions will incorporate such facilities in order to provide some of the proposed functions of higher levels.

The mark operation provides a tool which can be used...
by the higher levels of the system to implement complex relational operations (e.g., joins). A mark operation on a mark is also allowed. This results in the creation of a subset of indices of a first mark according to a second qualification. A mark on the schema is also allowed where the schema is viewed as a relation. This way the users can isolate the parts of the schema for high level security or for data dictionary queries.

The qualification for a mark is passed as a binary tree of a Boolean expression of basic domain-comparator-value triplets. A method for efficiently executing Boolean qualifications on partially inverted relations has been studied and is being incorporated into the system.

THE INTERMEDIATE RELATIONAL LEVEL

This level is concerned with supporting high level relational data base structures. Two important functions are provided at this level: derived relations and multi-relation queries. It provides the higher language level an interface which is suited to its user’s goals and interfaces with the basic level in order to accomplish its tasks. The interface to this level is through a rigid data structure so that interpretation is straightforward.

The execution portion of this level must concern itself with the semantics of user requests with respect to the method used to implement relations. Relations that have been implemented as virtual constructs require different algorithms for executing the various types of operations that can be performed on them. Also, high-level optimization can occur whenever an operation or relation implementation can be accomplished by more than one method. The intermediate level is responsible for these decisions.

Derived relations

A primary relation is one that has been created by the data base administrator. A derived relation is a relation that is formed as the result of a join of two or more derived or primary relations or the result of a qualification giving a subset of a single relation’s domains and tuples. Derived relations are an important component of a DBMS because users often need focus on a subset of the data base. More important, the re-execution of similar queries causing extensive retrievals are very expensive. ZETA allows a user to name and define a new relation with the same constructs used for qualifying and retrieving from an existing relation. Two types of derived relations play an important role in this system: snapshots and automatic derivations.

A snapshot is a time invariant “picture” of a portion of the data base at a particular instance. The resulting relation is no longer dependent on changes made to any of the relations involved in its creation. Such derivations are important as a journaling facility or because a stable environment is required. Such is required by programmable DBMS where operations tangential to retrievals are occurring to the data of a relation. If a snapshot were not created, then the original relation would have to be locked from updates perhaps for an uncomfortable amount of time.

Several implementation schemes have been considered for snapshots which define a subset of a single existing relation. It became clear that an optimal one depends upon the type of system usage that is encountered.

In a retrieval oriented environment (i.e., update is batched and performed when no retrieval is occurring), the marking scheme presented by the primitive level is perfectly adequate. An access to a derived relation in this configuration would require the indirect access through a pointer to a tuple. Since the relation is not stored as an “actual” relation it is called “virtual”.

In a very active update environment, the great amount of overhead incurred by updating a pure marking scheme would make it prohibitive. Instead a snapshot could be formed by creating a separate actual relation—one whose tuples are the records of a separate file. This is accomplished by first locking update access to the relation, then by forming a marking on the qualified tuples of that relation, and finally by retrieving and copying the required domain values to the snapshot.

Both of the above schemes are being implemented. However, since a straight marking is not adequate in an update environment and because complete actual implementation is costly in terms of storage, a partially virtual compromise is proposed. A snapshot is originally implemented as a marking. Each time a change occurs to a tuple of the original relation, the old tuple is copied and logged into a separate section of the primary relations file. The mark that originally pointed to the actual tuple now points to its new location and the original tuple is updated. When the number of marks which points to the augmented primary relation reaches a threshold, then the snapshot is converted to an actual representation and the marks are destroyed.

An automatic derivation is a time-dependent relation which reflects the changes that have occurred to each of its ancestors. A snapshot would have to be re-created each time a user wants an updated version while an automatic relation remains “up-to-date”. Several schemes have been proposed.

An automatic relation is equivalent to a snapshot in a retrieval-oriented environment. Because of this, a marking is completely adequate. However, in this case the problem is of no interest because, by definition, updates are why we consider automatic derivations.

One simple scheme would be to save only the definition of the relation. When a query involving the derived relation is initiated, the relation is locked against other disturbances, the derived relation is formed from the stored definition, and then the relation is destroyed. This is, of course, an expensive procedure but may prove adequate in an environment where automatic relations are required only once in a while.

A third, intermediate scheme, conforms to the same philosophy as the partially virtual scheme presented for snapshots. At first, the derived relation is formed as a
marking and its definition is saved. This time, changes are logged by storing the new tuples as part of an augmented primary relation and marks pointing to the old tuples are made to point to the new tuples. These changes must be time stamped. The next time the derived relation is accessed, the definition is re-executed only on the tuples not checked since the last access.

It is clear from this discussion that derived relations must be used carefully. Snapshots must not be kept too long and automatic relations must not be used too often. Also, the incorporation of derivation schemes requires the formation of an intermediate level schema for the translation between logical operations and virtual implementations.

**Multi-relation queries**

A multi-relation operation or query requires the manipulation of data from two or more relations which can be implemented either as primary or derived relations. Two types of multi-relation queries arise in this system: join and composite.

The join operation creates a new relation by combining two existing relations. This is accomplished by comparing the relations on one or more domains that are common to each relation and by forming the new tuples by associating a tuple from one relation with one or more tuples of the other.

The composite operation (also called an implicit join or restriction) extends the qualification capability for selecting subsets of tuples of a relation. It allows the results or values retrieved from one relation to be used as qualification for the retrieval over a second relation.

**System structure**

The intermediate level system is composed of three principal components: the interpreter, a set of intermediate level schema procedures, and a set of utilities. All three components are embedded within a single external procedure in which the intermediate level schema tables and interface command data structures are global. This system is invoked by a call from the higher level system passing the command data structure as a parameter. The interpreter breaks the command down into a sequence of utility operations. The utility procedures interface with the primitive level system. Schema procedures are invoked by the interpreter whenever data pertaining to derived or primary relations are to be stored or retrieved.

**THE USER INTERFACE LEVEL**

Three major classes of users requiring access to DBMS can be described: application programmers, technical personnel, and casual users.

An application programmer requires a programmable DBMS language in order to access data to formulate complex queries, special reports, or perform atypical computation not provided by other interfaces. He must understand the organization of data but can often ignore the meaning of the data being processed.

A technical user is a person who specializes in the semantics of the data and does not want to concern himself with computer system details. Examples of such users are urban planners, doctors, administrators, management scientists, engineers, etc. Query languages have evolved to provide these technical users with an English-like language for interactive communication with a DBMS. The power of this type of interface is that the user does not have to seek an application programmer for most of his data processing and that he can “browse” through his data base by reformulating queries based on data just received.

The casual user is a person who either requires access to a DBMS so infrequently as not to warrant learning a high or low level programming language or who refuses to learn the discipline of a query language. The only dialogue acceptable is some form of natural language.

All three types of interfaces have been investigated for interacting with the lower levels of the relational DBMS.

**A programming language interface**

A programmable interface to this relational DBMS is being implemented by embedding a set of data manipulation language (DML) subroutines within a PL/1 host language environment. The syntax for a qualification is a modified subset of SEQUEL. Commands to the DBMS are passed as structured English-like character strings within a procedure call. A preprocessor has been designed so that a “cleaner” interface can be effected and so that syntactic errors can be flagged before execution of the PL/1 program.

The DML embodies the capabilities of the intermediate level system, thus making the translation process straightforward. Its capabilities provide:

(a) Multi-relation qualifications,
(b) Derived relations based on qualifications,
(c) Updates based on qualifications,
(d) Interaction with host program variables in order to express complex queries by:

(i) passing “generated” values for dynamic qualifications,
(ii) passing “generated” requests.

Since this language has all the capabilities of PL/1, it could potentially serve as a system implementation language for the development of language processors of the following two types.

**A query language facility**

The most important characteristic of a query language is how well it is tailored to its problem area. General pur-
pose languages often force users to think in terms of unnatural logical data structures and syntax. The goal of this work is to construct an environment in which problem-oriented query language generation could be readily accomplished.

An existing syntax-table-driven compiler-compiler was modified to support a query language development facility. This system allows users to easily specify the syntax of their language and to connect it to a set of modular semantic routines. The semantic routines utilize the intermediate level (which is accessible through PL/1) as a semantic language. This allows the symbol table and other system tables (which could include the tables of the syntactic network used by the fixed parser) to be stored, retrieved, and maintained by the DBMS.

Because the syntax of a users source language is specified in a table, subsets of a language can easily be generated by eliminating parts of the table. This provides several benefits:

(a) A customized user view; unused capabilities may be omitted,
(b) Security; e.g., by eliminating the UPDATE keyword for a specific user, the associated capability is unaccessible,
(c) Reduced compilation time; the number of optional syntactic constructs are reduced so that parsing time is shortened.

A special facility for macro definition and expansion was also added to the generator. The use of macros in a query language can provide the powerful capability of allowing one user to specify a complex query and have another utilize it easily. It also provides a much needed abbreviation facility for teletype terminal environments.

Because a specific application data base has not yet been explored, a general purpose relational query language was built using the facility. It embodied all the features of the intermediate level system and added:

(a) A sequence of language subsets,
(b) User macro definition,
(c) Report generation:
   (i) sorting,
   (ii) output device choice,
   (iii) titles and headings,
   (iv) position and type formatting,
(d) Data dictionary.

An intelligent natural language interface

The expected increase in the use of DBMSs will cause an ever increasing need for making DBMSs more readily available to the casual user. The aim of the TORUS (Toronto Understanding System) project is to achieve this goal by:

(a) Eliminating the need for the user to know how information is stored in the DBMS and only expecting him to know what is stored,
(b) Enabling him to communicate with the DBMS in simple English and without the arbitrary restrictions of programming languages.

The methodology we have chosen in order to achieve the TORUS aim is different from that of earlier attempts to provide a natural language interface with the user. TORUS extensively uses semantic networks as a basis for "understanding" the ongoing dialog with a user as well as the information stored in the DBMS. This methodology was chosen because it is compatible with the philosophy that there are no shortcuts to the problem of making a system appear intelligent to a user who lacks the training and/or the time to program it. This intelligence can only be realized by a system which has and can use the knowledge relevant to the universe of discourse—in this case a data base—in ways which, at least abstractly, are similar to those used by humans. It follows from these premises that the representation and use of knowledge about a particular data base stored in a relational manner is the most important problem to be tackled in designing a natural language front end to a DBMS.

TORUS consists of four basic modules:

(a) INPUT: The input module accepts a sentence in English, performs a syntactic analysis, and outputs a tree structure which describes the sentence's underlying syntax. This structure is then passed to the SEMANTICS modules.
(b) SEMANTICS: This module is responsible for "making sense" out of the structure passed by INPUT. This is achieved by using a semantic network and associated procedures, and results in a complete integration of the input sentence with the semantic network. If the input structure does not make sense, another parse is requested from INPUT. If there is a need for use of the DBMS, appropriate commands are passed to the INTERFACE module and execution of SEMANTICS is suspended until a message is returned by the interface. It is then decided what should be output and a structure similar to that constructed by INPUT is sent to the OUTPUT module.
(c) OUTPUT: The structure received from SEMANTICS is mapped on to a sentence, using some previously explored ideas.
(d) INTERFACE: This module receives lists of commands to update, retrieve, or test the validity of information in the data base. These lists are translated into sequences of commands for the DBMS and its responses are passed back to SEMANTICS.

The system is currently being tested with a data base involving student records. Each record stores general in-
formation about a student (place and date of birth, student number, marital status, etc.), his/her academic background (universities attended, courses completed, degrees obtained) and his/her record in the Department of Computer Science at the University of Toronto.

The functions of the system's modules will be illustrated by considering a simple sentence like "What is John Smith's address?" and describing how it will be handled by the TORUS system.

When INPUT is presented with the sentence, it first performs a lexical analysis of the words present. 'John' and 'Smith' have to be marked as proper nouns by the user who types in the question. The sentence is then passed to an augmented transition network parser which attempts to find a parse. It must be noted that the parser will not attempt to resolve semantic ambiguity problems (e.g., reference of prepositional phrases, selection of a word sense when there are several candidates in the same syntactic class as in "I took a course" and "I took a book", etc.).

The structure constructed by INPUT is then passed to the semantic module, which first attempts to construct a graph representing the meaning of the input sentence. This is done by first using "case frames" which check whether the semantic relationships implied by the parsed structure are meaningful. Several "mapping functions" are then used to obtain a deeper representation of the meaning of the input sentence, e.g., 'take' may have a mapping function which will map the structure described by:

\[
\text{[take: agent=student, object=course, source=professor]}
\]

which may have been obtained from "The student took a course from the professor" into the semantically deeper structure

\[
\text{[teach: agent=professor, object=course]}
\]

\[
\text{[enrol: agent=student, destination=course]}
\]

There is no need for mapping functions for the sample sentence.

Anaphoric reference problems are also treated at this point. The function which attempt to solve such problems will take into account semantic properties of the object referred to and will attempt to identify it on that basis.

Another job that has to be done at this time is to determine the type of action (i.e., retrieval, update, count, test-for-validity) that will have to be performed by the system. In this case the action is 'retrieve'. Note that the sentence "Give me John Smith's address" will be mapped into the same graph as that of our current example, 'give' serving only as an indicator that it is the 'retrieve' command that is applicable here.

The next step involves "fitting" the constructed graph to the semantic network which stores general knowledge the system has about student records as well as specific information it has obtained during the current conversation either from the data base or the user. In the network there will be nodes (concepts) which correspond to our general notion of 'address' (it is something that can characterize a person or an organization) as well as that of 'current-address'. Moreover, 'current-address' will be represented as a sub-concept of 'address' and it will be associated with an attribute of the relational data base named CURRENT-ADDRESS where a student's current address is stored. Thus the semantics of the attribute CURRENT-ADDRESS of the relational data base is defined by the properties of the concept 'current-address' of the semantic network.

During the fitting of the graph, "recognition functions" are often used to determine whether a node of that graph represents an instantiation of a generic concept on the semantic network. A recognition function can use common-sense knowledge about a generic concept which might help it decide that 'abc def' is not an address, but it may also check the input sentence and the semantic network to make inferences about which nodes of the input graph are instantiations of a particular generic concept.

Once the graph has been fitted on the network, the system knows that something must be retrieved from the data base (since a 'retrieve' command was generated) and where to find it (since the 'address' node of the input graph was found to be an instantiation of the 'current-address' concept to which the attribute CURRENT-ADDRESS is associated). A message is therefore passed to the INTERFACE.

\[
\text{(GET 1; ADDRESS=?; NAME=John Smith)}
\]

which asks for the retrieval of a single item from the data base that is consistent with the information given (the name) and includes the missing information (the address). The interface will return

\[
\text{(ADDRESS=65 Charles St., Toronto; NAME=John Smith)}
\]

and SEMANTICS will then place this information on the semantic network.

Finally, SEMANTICS decides what portion of the semantic network should be output by taking into account the input sentence and the information that was retrieved from the data base. The structure constructed is quite similar to that created by INPUT, except that the question mark has been replaced by '65 Charles St., Toronto'.

Several "inverse-mapping functions" may be used during the construction of this structure, mapping the deeper representation of the sentence to be output into a more surface form. The OUTPUT module now constructs a string by using an augmented transition network.

Several features we consider important have been implemented superficially in the current version, and others are missing completely. Thus the system only handles simple cases of conjunction, disjunction and quantification. Moreover, it has limited inference capabilities and its "understanding" of the ongoing dialog is accordingly restricted.
CONCLUSION

The ZETA data base management system is implemented in PL/1. The primitive level is completely designed, implemented and tested. The intermediate level and query system level has been designed and partially implemented. We are finishing implementation and integration at this point. A report with the status of the project is forthcoming.

A first version of TORUS, which handles simple sentences and contains only a few mapping, inverse mapping, and recognition functions has been implemented and is currently being tested. The languages used for the implementation are SPITBOL20 and 1.PAK an AI language offering directed labelled graphs and graph pattern matching.21 The details of the current version can be found in a technical report.22

ACKNOWLEDGMENT

We would like to acknowledge the contributions of the following students and colleagues who made the ZETA/TORUS project a reality: Michael Brodie, Simone Chan, Bryon Czarnik, Gilles Farley, Jack Klebanoff, Edwin Leong, Ellen Poon, Olga Puchmajerova, Dick Swenson, AIda Usas for ZETA; Alex Borgida, Phil Cohen, Nick Roussopoulos, John Tsotsos, Lindsay Watt and Harry Wong for TORUS.

This research was supported in part by the National Research Council of Canada, the Department of Communications of Canada, and by IBM Canada Ltd.

REFERENCES

19. Martin, W. A., Menos 6, 8, 11, 12, 13, Automatic Programming Group, MIT.