CAPTURING MORE MEANING IN DATABASES

David Maier*

Department of Computer Science and Engineering
Oregon Graduate Center
Beaverton, OR 97006

Technical Report #CS/E 83/009

1. Introduction

There is a component of every database system that is seldom formalized and even more rarely explicitly supported by the database system. This component is the meaning of the data in the database. This paper examines the problem of conveying the meaning of the database to humans and programs that use the database. We shall examine some current efforts in this area, to see what the next generation of commercial database systems might offer and what we might expect five years from now. We shall also point out directions further research might take.

In our discussion we shall draw examples from a database designed for recording energy production, flow and consumption in the United States [EEMIS]. What we intend by the meaning of a database is the information that the user needs to make sense of the data in the database. (This is a minimal definition of database meaning--certainly there can be aspects of a database's semantics that lie outside this definition.) It is the answers to such questions as:

What does the value "5OR6" in the ENERGY-TYPE field stand for?
What objects in the database could correspond to refineries in Oklahoma?
What does an instance of the OIL-SHIPMENT record represent?

How should we convey the answers to these questions to the user?

In many existing database applications--an inventory, for example--the scheme of the database is quite simple. There is a small number of fields,

* This work was supported by NSF grant IST 79 18264 and IST 81 04834.

and the meanings of the fields can be learned in minutes, or may even be obvious from the field names. Record instances are likely to be uniform. In other applications, or potential applications, the scheme can be large, the meaning complex, and record instances can exhibit wide variations. The record format for an oil refinery could have over a hundred fields and instances of it may differ because some refineries do only distillation, while others also do cracking, or one record instance might represent a single refinery, while another represents the aggregate of all refineries in a state.

A user or application programmer must know the structure and meaning of a database before he or she can utilize it. For a simple database, this knowledge can be acquired through a ten-minute chat with another user. For a complex database, a user might not even be able to remember the entire database structure at once. There is also the danger of oversimplification in complex applications. Certain variants or details may be elided because they complicate the database scheme. The database system should assist the user in storing, recalling, and manipulating the information necessary to access and make sense of the data in the database. The database must somehow capture its own meaning.

To better understand this problem, we shall first look at the parallels between databases and programming languages. We next examine several approaches to the precise specification of conceptual data models and the means of describing the connection between a specific conceptual model and the database scheme that implements it. We explore the feasibility and advantages of storing a semantic description of a database as part of the database itself. Finally, we briefly present an experimental database query language, PIQUE, that uses such information.

2. The Analogy to Programming Languages

In this section we try to point out some similarities between programming languages and databases. The analogy is tenuous and inexact at times; it is not intended to be carried further than it is here. Figure 1 diagrams the basic task in both areas, which is one of translation. With programming languages, the task is to translate from abstract concepts of algorithms, such as "Bucket sort" or "greedy algorithm for memory allocation," down to machine code programs. With databases, the translation is from mental models of real-world information, such as "inventory of parts" or "employee personnel records," down to configuration of bits in memory.

* * * * *

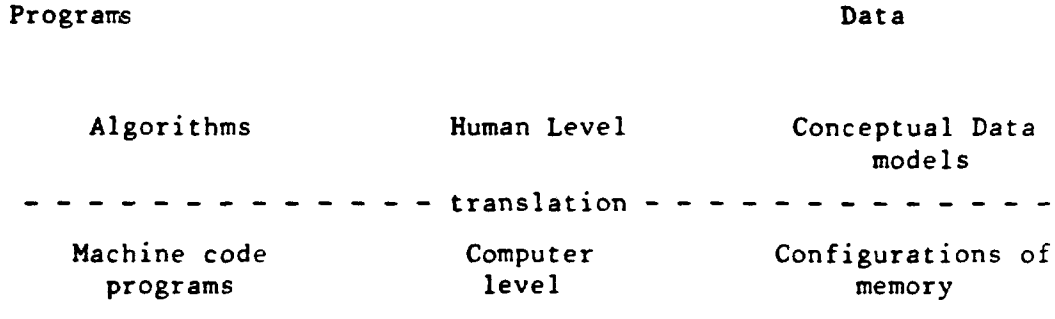

Figure 1

* * * * *

In either case, the computer level is at too low a level of detail to easily comprehend, and the organization is not at all similar to the organization we use at the human level. In the programming languages domain, we have assembly languages and intermediate codes as representations that are one step up from machine code programs. For databases, we have data types as ways to view memory configurations. We shall call the level of assembly language, intermediate code, and data types the structural level, for it is mainly concerned with structuring, grouping and naming objects at the computer level (see Figure 2).

* * * * * *

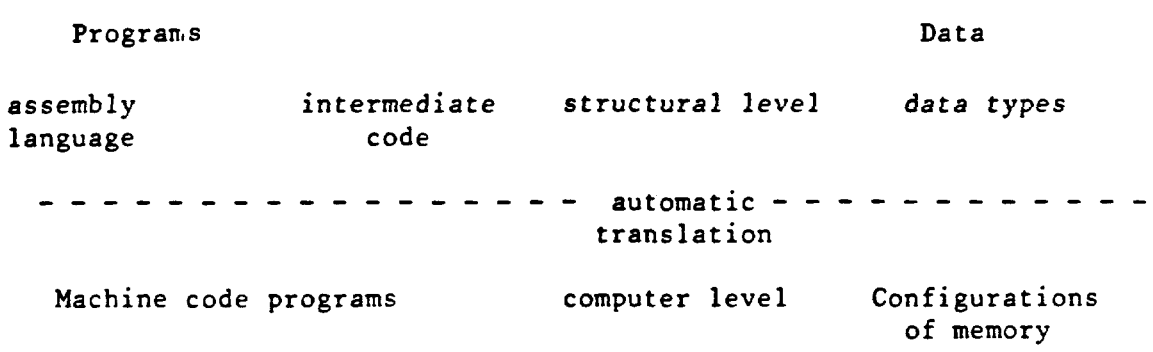

Figure 2

* * * * * *

In all cases, we have automatic translation from the structural level to the computer level. There are assemblers for assembly language, machine code generators in compilers for intermediate code, and standard implementations in programming languages for data types, as well as operations for manipulating the types. For example, most programming languages have matrices, and many support the matrix data type. (That is, they include operators, such as matrix-multiply and transpose, whose arguments are entire matrices.)

Admittedly, there is more abstraction on the data side than the program side in going from the computer level to the structural level. In particular, for databases we have very abstract data types (such as relations and hierarchies), which are usually called data models. The most important aspect of the structural level is not the amount of abstraction layered over the machine, but that the abstraction is machine oriented. Even the relational data model, which is far removed from the bit level, was intended to mask the complexities of file and index manipulations, rather than to provide a natural framework for modeling real-world information.

The advantages of working at the structural level are manifold. There is a simplified view of the computer: many details are taken care of automatically or suppressed, reducing the choices that must be made, and the opportunities to go astray. There is an opportunity to incorporate mnemonic

information through the naming of instructions, locations, variables and so forth. Optimization seems most easily done at the structural level. And, in the case of intermediate code and data types, we are offered some physical independence. If we want compilers for a single programming language for ten different machines, we do not have to write ten compilers from scratch. A single complier can be written through the intermediate code stage and then fitted with various machine code generators, one for each of the ten machines. The implementation of a particular data type can be changed to account for different machine capabilities, such as different word sizes or different instructions.

Despite the advantages of the structural level, the languages and constructs there are essentially a means for looking down at the lower level of detail of the computer. The representations at the structural level still do not correspond that well to the concepts we use at the human level. They do not help that much in translating from a mental idea to a more formal and precise specification of an algorithm or database--we must still go from human terms to computer terms, although the distance is not quite so great.

What we want are mechanisms for looking up to the human level. We want a method for precise and formal specification of algorithms and databases that nevertheless allows us to understand what is going on in human terms. We want precise definition with easy understanding. At the structural level we do have naming and comments to help us understand assembly language programs, but in a sense they do not count. The assembler does not attach any meaning to the names, nor does it read the comments. The intent of the comments are not guaranteed to be carried out in the programs. Mnemonic names for instructions do help somewhat, because they are easier to read than machine code and we are given a precise translation.

Let us examine mechanisms for looking upward toward the human level. Most of us who do programming or program design have some semi-formal notation in which we work out the initial design of a program, such as structure charts or pseudocode. Such notations are precise enough to serve as a useful tool for discussion and evaluation of a design. Furthermore, we can establish a correspondence between boxes in the structure charts or pseudocode statements and groups of statements in the actual program once it is written. For example, the pseudocode statement

let MAX be the largest value in array A

might correspond to the program segment

```
MAX := A[1];
FOR I := 1 to ALIM DO
     IF MAX < A[I] THEN MAX := A[I];
```

Thus, when there is a change in flow diagram or pseudocode, we can easily locate the corresponding part of the actual program that should change. Again, however, we have no guarantees that our intent in the flow diagram or pseudocode is carried out in the actual program.

We generally have some translation in mind for each chart box or pseudocode statement. A high-level programming language can be regarded as an attempt to formalize someone's pseudocode sufficiently to automatically translate it. It is a step up from the structural level.

We have evolved from BLZ to arithmetic IF and GO TO's to IF-THEN-ELSE and WHILE-DO.

In SmallTalk-80, we can define a unary operator ("message" in SmallTalk terms) 'max' that will select the maximum element from any type of collection that has a greater-than relation defined on its members. We approach more and more closely our own concepts and are tied less and less to the concepts of the machine. Of course, we trade the ability to easily specify some high-level programming constructs for the ability to specify arbitrary machine code programs. It becomes easier to specify a looping structure, but harder to manipulate individual bits of memory. But the trade pays off, since we need to manipulate individual bits of memory seldom compared to how often we need looping structures. We are creating a semantic level--one that tries to represent the meaning of programs in human terms, but still formally. We are ever decreasing the distance that a human translation has to traverse (see Figure 3).

* * * * * * * *

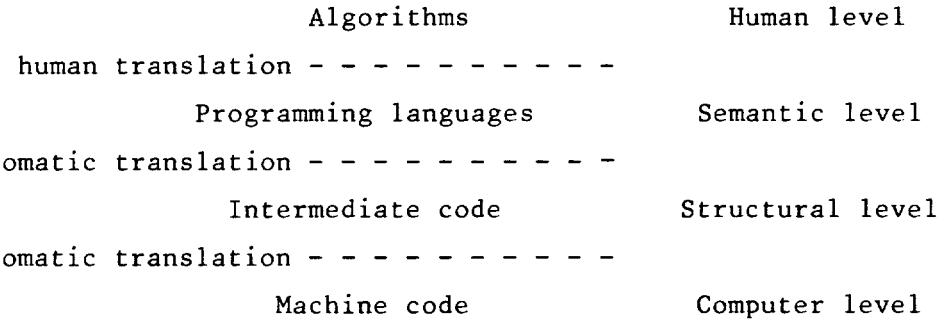

Figure 3

* * * * * * * *

What is the corresponding state of affairs for databases? The database models supported by current database management systems (DBMSs) are not much above the structural level. They are essentially data type packages provided to simplify our view of the computer level, but there is little correspondence between the objects in our conceptual model and structures in a database. For example, in the relational database model, everything is organized into tables. However, our conceptual models of data only correspond directly to tables in cases such as inventory lists and airplane schedules. We do have naming, but the name of a field or record does not affect the treatment the DBMS gives that field or record. There is still a long jump to the human level.

Semantic data models, as shown in Figure 4, are an attempt to make possible unambiguous descriptions of people's ideas of specific data applications. We discuss semantic data models in the next section. They have not reached the status of high-level programming languages; we are just beginning to see automatic translation from them to the structural level in prototype DBMSs [Ca,Ch+, We]. Nevertheless, they are useful for discussing the intended meaning of a database, and someday automatic translation will be available in commercial systems.

* * * * * *

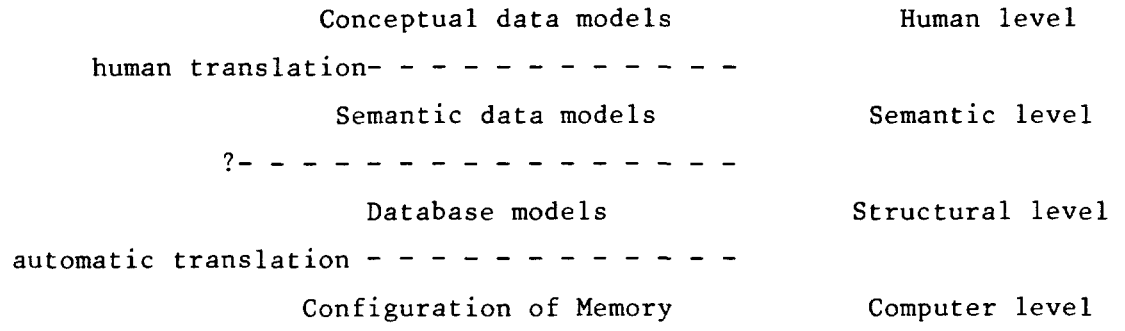

Figure 4

* * * * * *

3. Semantic Data Models

It is possible for one person to impart to another the general meaning of a database informally, but to discuss the faithfulness of an implementation or to store the meaning of a database as part of a database some formalism is necessary. Semantic data models are systems for precise specification of conceptual data models, but, unlike database models, they attempt to use terms and concepts humans use. We briefly cover four such models.

Probably the most well-known semantic data model is the entity-relationship model of Chen [Ch]. In the entity-relationship model there are four types of objects:

1. entities
2. attributes of entities
3. relationships among entities
4. attributes of relationships.

Information structures in this model are easily expressed through diagrams such as Figure 5. Rectangles denote sets of entities, diamonds are relationships and circles are attributes of entities or relationships. The entity-relationship

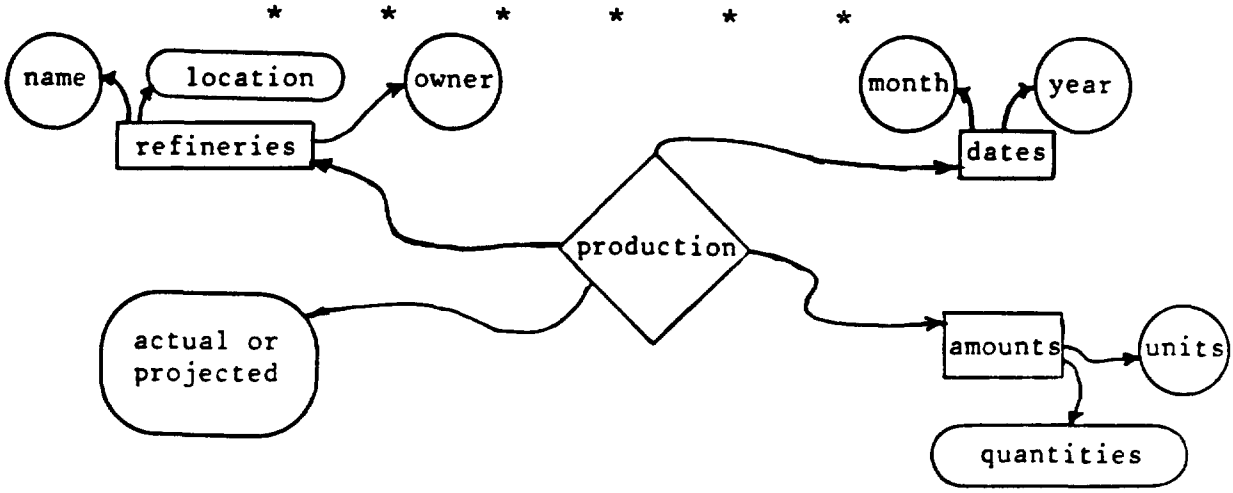

Figure 5

* * * * * *

model, as first presented, does not speak much to definition of domains of attributes or any involved integrity constraints. It is simple and readily understood, but does not capture the complexities that arise in certain database applications.

Brodie has defined a semantic data model through his language BETA [Br]. BETA focuses on definition of domains and integrity constraints on the database. BETA has objects and maps (roughly corresponding to entities with attributes and relationships in the entity-relationship model), and also assertions. One goal of BETA is to be able to make inferences from a database description and to be able to check the consistency of the assertions. A partial BETA specification follows:

```
const curyear = 83;
type
refinery = object name: nametype;
                  location: loctype;
                  owner:  company-type;
                  keys:  name, (owner, location);
           end object;
dates = object month: ∉(1..12);
                year: ∉(1960..1995);
        end object;
```

```
production-records = object facility:  refinery;
                             period:  dates;
                             ptype:  (actual, projected);
                     end object;
produced-amount = map from p in production-records
                      to a in amounts
                  end map;
assert:  all p in production-records
               where (p.date.year > curyear)
               (p.rtype = projected);
```

BETA also has features for describing sub-objects and sub-maps. BETA resembles a database definition language with the addition of assertions. It is slightly restricted in that maps are always binary relations.

Biller and Neuhold's Logical Data Definition Language is similar to the BETA language, except it deals in types and relations on types [BN].

The semantic data model SDM of Hammer and McLeod [HM] has classes, with members of classes having attributes, and various means for defining classes from other classes, such as restriction or grouping. For example, if we were interested in looking at projected production by itself, we might define

```
PROJECTED-PRODUCTION-RECORDS,
     interclass connection:   subclass of PRODUCTION-RECORDS
                              where PTYPE = "PROJECTED".
```

The last three semantic data models are still somewhat downward-looking. Specifications in each case come out looking like database schemes or data type declarations in a programming language. The problem is not wholly in the models; some of it lies in the terminology and syntax used for specifications in the model. "Interclass connection" is not an everyday term. Humans would more likely phrase the specification above as:

"A production record is either actual or projected."

without creating a name for the concept. Of course, natural language expression and precision are often antithetical. So, while these semantic data models are removed somewhat from the database scheme level, there is still a computer flavor.

Another manifestation of the "downward-looking" tendency of some semantic data models is their lack of modeling primitives for data manipulation; the problem Codd characterizes as "anatomy without physiology" [Co]. Even when the structure of the data can be modeled in high-level and near-human terms, manipulations on the data end up being expressed as "add element x to class y"

or "change property p of x to c." We would much rather express our updates in the form "x opened a new refinery in Oklahoma" and "refinery x closed its cracking operation." For more on the problems of modeling real-world actions in the database world, see Maier and Salveter [MS, SM].

It is fairly straightforward to design a database scheme that implements (but doesn't support directly) a data description expressed in the language of one of these semantic data models. All of the authors give design methodologies to go from a semantic specification in their model to a database scheme. It does not seem an impossible task, either, to build a DBMS that lets the user deal with the data in the database at the level of the semantic model. As mentioned, we are already seeing prototype systems of this type.

4. The Correspondence Between Semantic Specifications and Database Schemes

For a semantic data specification to be a useful device for describing the meaning of a database to a user, there must be a precise means of expressing the relationship between the semantic specification and its implementation in a particular DBMS. The correspondence may not always be transparent. For example, a database implementation would probably not have two distinct files for production records and the subset of projected production records. Rather, projected records would be distinguished by the value in some field of the production record. Also, the values "actual" and "projected" would not be represented as character strings, but as shorter values, such as 0 and 1. Company names would likely be abbreviated. These conventions must be part of any explanation of the meaning of the database.

A group formed at Stony Brook (which is now widely dispersed) has been working on a data semantics language (DSL) to be used to express the correspondence between semantic specifications and database schemes. The DSL will formalize such statements as

> Projected production record is a production record where ptype = 1.
>
> 0 is the database representation for "actual" in the ptype field.
>
> Closing an oil processing operation means updating the capability field in a refinery record.

The first use envisioned for DSL is expressing the correspondence between semantic data specifications and database schemes, once the correspondence is established. A longer range goal is developing a system that does automatic or computer-aided translation from a semantic model to schemas in a particular DBMS and produces a DSL description of the correspondence. The final goal is to then let the user deal with the database through the semantic model rather than through the DBMS implementation, and also allow the user to ask questions of the DBMS about the structure and meaning of a database.

This last goal has been partially realized by Harris' ROBOT system (commercially marketed as Intellect) [Ha]. ROBOT is a natural language front-end

for existing DBMS's. The semantic model is the English language, although the semantic specifications for databases are not explicitly incorporated. Users are assumed to know the English description of the information in the database.

ROBOT translates English queries to a precise English-like format and thence to the retrieval language of the given DBMS. Some examples (with slight typographical liberties):

GIVE ME THE NAME OF ALL EMPLOYEES WHO HAVE JOBS WORKING AS A SECRETARY IN CHICAGO.

--Print the NAME of any EMPLOYEE with CITY = CHICAGO and JOB = SECRETARY.

ROBOT gives the same translation for

PRINT THE NAMES OF ALL CHICAGO EMPLOYEES WORKING AS SECRETARIES.

and

WHO ARE THE CHICAGO SECRETARIES?

Other examples:

WHO EARNS BETWEEN $20,000 and $30,000?

--Print the NAME of any EMPLOYEE with SALARY between 20000 and 30000.

LOS ANGELES AREA MANAGERS?

--Print the NAME of any EMPLOYEE with JOB = AREA MANAGER and CITY = LOS ANGELES.

ROBOT's knowledge of the general structure of the English language is contained in its parsing routines. This component does not change from application to application. The correspondence between the semantic level and a given database implementation is kept in a dictionary that gives English words and phrases and corresponding database structures, referents for wh-words (who, where, ...), abbreviations for words, words for abbreviations, and so forth. In addition, ROBOT regards values in the database as being in the dictionary, so it is not necessary to enter every possible name as part of the dictionary.

ROBOT cannot answer questions about the semantic description, since ROBOT has no information on it. ROBOT draws a blank on

WHAT IS THE MAXIMUM NUMBER OF EMPLOYEES ALLOWED TO BE ASSIGNED TO A SINGLE MANAGER?

ROBOT cannot answer questions about the correspondence between semantic specification and database, even though it has the information, because the dictionary is not treated as part of the database. The query

WHAT IS MA AN ABBREVATION FOR?

will also come up empty-handed.

5. Storing the Database Description

We have seen that for a user to utilize a database, he or she must know, in addition to the database query language,

1. the semantic description of the data represented in the databse,
2. the structure of the database, and
3. the correspondence between the two.

For example,

1. production records can be actual or projected,
2. there is a PTYPE field in the file PROD-RECORDS, and
3. a 0 in the PTYPE field means actual, a 1 means projected.

One could argue that not every user needs all these kinds of knowledge. For example, a database administrator may be able to provide views for casual users that embody the correspondence between semantic and structural levels. For example, a view could replace short, stored values (0 and 1) by longer, more meaningful values("actual" and "projected"). This meta-information is spread among a number of sources, with possible duplication and contradiction, in most current information systems:

S1. the database itself
S2. system files for the DBMS that are maintained outside the database
S3. application programs that use the database
S4. external documentation
S5. people's heads.

Source S1 seldom contains more than structural information about the database, such as field names for each record. Source S2 is the database sheme given in the data definition language of the DBMS, although a compiled form of the data definition may reside within the database. Source S3 can contain many types of information about the meaning of the database, although in implicit form. For example, a report generation program may know to translate the values 0 and 1 in the PTYPE field to "actual" and "projected" in the output. Source S4 includes users' manuals, system specifications and operator's instructions. Source S5 can contain any of the above items and more.

5.1. How and Where to Store the Database Description

If descriptive information is to be stored in one place, the obvious questions are Where? and In what form? In answering these questions, we must be mindful of who or what will be using the information. Will it be new users, old users, application programs or some combination of the three?

Researchers in artificial intelligence are using semantic networks to represent general knowledge. Such networks contain nodes that represent objects, classes of objects and concepts, and labeled links between nodes that represent connections between objects and concepts. Semantic network research has much to say about methods for storing descriptive information for databases, but semantic nets do have two drawbacks. The information in semantic networks is not meant for human consumption, at least not directly.

Programs must traverse the network and manipulate the information before a human can make sense of it. Semantic networks also allow more generality than is needed. Since we know in advance the types of information we plan to store, we can hope to find faster and more concise systems than one that must be prepared to deal with any kind of knowledge.

The method for accessing the database description is important; the description must be readily available to human and program users. We would like users not to have to learn another access method besides the database query language. Some database systems allow access to system files containing the database structure, but not with the same access method used for the rest of the database. A user has to learn special commands to get at this information.

We propose storing all the descriptive information in a standard format as part of each database. We then garner the benefits of uniformity of access and of having the information in one place, available to humans and programs. Users need only learn the format of the database description once to learn the descriptions of many databases. The range of possiblities for database independent programs--programs that will run on more than one database--is extended. Even very special purpose programs are less subject to change with changes in the database structure. Some existing database systems do store a small amount of this descriptive information as part of the database. For example, Query-by-example [Z1] and INGRES [St+] have standard relations giving the relation names and attributes in each relation for every relation in the database. Another approach, which is being used with increasing frequency in integrated information systems, is to provide a data dictionary manager along with the other system programs such as report writers, data entry programs and browsers. The dictionary manager keeps track of the structure and status of data in all parts of the system: the database schema, input files, data files outside the database, on-line manuals, and possibly data not in machine form, such as documentation and data entry forms.

The descriptive information should be stored using the same data structures as the rest of the database. Having different data structures for this information would mean the added compelxity of essentially a second database system to manipulate and manage the descriptive component. In addition, if the description information looks like part of the regular database, it can describe itself. Of course, update access to the descriptive component of a database must be carefully restricted, especially if system programs operate off this data.

5.2. Uses for the Descriptive Component

Once the description of the database is incorporated as a component of the database, to what use can it be put? Having this information available in standard format will allow a user to answer many of his or her questions about various databases without having to consult sources outside the database. More interesting are the types of application and system programs that could be written to take advantage of the descriptive information. We give some categories of applications below, and present a specific program in the next section.

Database Tutor: A program to introduce a new user to a given database and provide help interactively for all users.

Natural Language Query System: A system, such as ROBOT, to translate English questions into database query language. Unlike ROBOT, the query system would not have a separate dictionary. All the information it needs will be part of the database. Thus, the system can handle "meta-questions" about the description of the database.

Loaded Query Detector: Answers to queries, although literally correct, can often be misleading. A response of 0 to "Number of production records for New Jersey 3 refinery" could be due to there being no New Jersey 3 refinery, as well as there being no production records if the refinery does indees exist in the database. A program looking at the descriptive information could detect cases where there are implied assumptions in a query that may not hold in a given case. Additional information could be given along with the flat answer if the implicit assumptions are violated.

Enforcing Constraints: The descriptive information is sure to include constraints on the state of the database. The database system should enforce these constraints and point out which constraints are violated when an update to the database is disallowed. Current systems can, for the most part, only handle constraints involving ranges for field values and keys for record instances.

Dealing with Partial and Unreliable Data: While constraints on the database are restrictions on updates, they can also be viewed as general statements about what should be true about a state of the database. As such, they can be used to fill in missing values in the database. For example, suppose each refinery has a unique owner. If there is one refinery record instance with a refinery name and owner, and a second instance with the same refinery name, but missing the owner information, the former instance can be used to fill in the latter. The database can also contain descriptive information for an individual data item, a data pedigree, source and reliability.

5.3. What Should the Descriptive Component Contain?

The hard thing is not deciding whether to include a descriptive component in a database, but deciding which information about the database to include in the description. We won't take space here to enumerate all the types of descriptive information that might be included, as the profitibility of including any specific type of information depends on the anticipated users, the data model and what applications are planned. However, we will offer one framework for developing the descriptive information, based on dividing the information into

1. semantic
2. syntactic, and
3. correspondence

categories for each construct in the data model. Below we give some examples of types of descriptive information for constructs in the relational data model: domains, attributes, tuples, relations and databases.

A. Domains: Sets of values representing entities and properties in the real world.

1. Definition of a class of entities: A fuel group is a collection of related petroleum products.

2. The format of domain values: Values in the FUEL_GROUP domain are 4-character strings such as AVIA, DIES, 5OR6.

3. The entities represented by domain values: AVIA is aviation fuel, DIES is diesel oil, 5OR6 is No. 5 or No. 6 fuel oil.

B. Attributes: The names of components in a record (field names in a record).

1. The intent of an attribute: Every storage facility has one or more storage capabilities, that is, the ability to store petroleum products of a given fuel group.

2. The domain of values associated with an attribute: FUEL_GROUP is the domain of attribute Storage_Capability.

3. What a particular entity or property represents as a value for an attribute: G as a value for Storage_Capability mean the storage facility means the facility has tanks for storing petroleum products in fuel group G.

C. Tuples: Lists of values corresponding to a list of attributes.

1. The connection among a set of entity classes: A storage facility a storage capacity for each of its storage capabilities.

2. The scheme of a tuple: There are tuples over scheme ⟨Storage_Facility, Storage_Capability, Capacity⟩.

3. The interpretation of a tuple in terms of its components: A tuple ⟨f, g, c⟩ over scheme ⟨Storage_Facility, Storage_Capability, Capacity⟩ means that storage facility f can store c barrels of petroleum products of fuel group g.

D. Relations: Sets of tuples over the same scheme.

1. A description of a set of related facts: For all storage facilities, there is total capacity and free capacity.

2. The scheme of a relation: Relations total_cap and free_cap have scheme ⟨Storage_Facility, Storage_Capability, Capacity⟩.

3. What set of facts a relation represents: Relation total_cap represents total storage capacities for storage facilities, while free_cap represents the unused storage capacity at the beginning of the current month.

E. Database: A collection of related relations.

1. Relationships between classes of entities: Every storage facility is an energy facility.

2. Inter-relational constraints: Every value in the Storage_Facility column of the total_cap relation must appear in the Energy_Facility column of the facility_owner relation.

3. Methods for inferring facts from the database: Owners of storage facilities can be related to storage capacities at their facilities by joining total_cap with facility_owner on Storage_Facility = Energy_Facility.

The list above is far from exhaustive, and it should be apparent that classification of a type of information into semantic, syntactic or correspondence is somewhat fuzzy.

6. The PIQUE Query Language

We describe here a database access program that is driven off meta-information stored about a database. PIQUE is a high-level query language for relational database systems [M+, Ro]. In addition to the usual relation scheme and domain definitions, it uses information about likely connections among attributes, captured as descriptors called associations and objects [MW]. The goal of PIQUE is to provide a query language in which the entities involved in a query all appear in the corresponding English query. That is, no purely database-oriented artifacts need appear in a query. PIQUE can be considered a "database back-end to natural language interfaces" (contrasted to "natural language front-ends for databases"). PIQUE is similar to tuple calculus languages, but it removes the need for binding tuple variables, and frequently eliminates explicit tuple variables altogether.

Consider an example database of three relations

```
Teaches(INST, COURSE#, SEMESTER)
Enrol(STUDENT, COURSE#, SEMESTER)
NAMES(COURSE, CNAME)
```

giving the instructor for a course each semester, the students in a course each semester, and the course names corresponding to course numbers. Suppose we want to answer the question

"Which students have taken Compiler Design from Warren?"

In QUEL, the query to answer this question is

```
range of t is Teaches
range of e is Enrol
range of n is Names
retrieve e.STUDENT
where t.INST = "Warren" and n.CNAME = "Compiler Design" and
      t.COURSE# = n.COURSE# and t.COURSE# = e.COURSE# and
      t.SEMESTER = e.SEMESTER
```

Note that certain database objects that did not appear in the English question crop up in the query, namely, the relation names and tuple variables. Also, the attributes COURSE# and SEMESTER have to be included to make the appropriate connections. This question can be answered with the PIQUE query

```
retrieve STUDENT
where INST = "Warren" and CNAME = "Compiler Design"
```

Admittedly, there still are two attribute names, INST and CNAME, that do not appear in the English question, but the database-oriented objects have been removed. Another way to view PIQUE is as a query language based on second-order, rather than first-order, predicate logic. A PIQUE query is implicitly quantifying over all stored and virtual relations that contain certain attributes. This quantification makes PIQUE queries partially immune to changes in the logical structure of a database.

Our current work involves extending the types of meta-information we capture about a database, and developing companion languages for updates and data definition. We are looking at ways of incorporating role (ISA) hierarchy information into PIQUE, and the possibility of adding stative verbs to replace relation names [MRS]. We are formalizing the process of automatically computing the connection among a set of attributes. We are also investigating mechanisms for supporting virtual relation definition that are more powerful than relational algebra, which cannot express all functions of relations. Logic programming languages are our most attractive alternative at the moment.

7. Conclusion

We have seen how development in databases is following that of programming languages in trying to come to more human terms. We examined several methods for precise semantic descriptions of data applications, and discussed the problem of representing the correspondence between such a semantic description and the database scheme that implements it. We proposed storing the database scheme semantic description and the correspondence between the two as part of the database and the benefits that may accrue from this approach.

We believe that it is useful to maintain the distinction between semantic and structural levels in database systems. Models at the structural level should be kept simple, with a small number of different constructs. The structural level should be geared toward performance, looking at both efficient implementation and keeping system overhead low. The structural level also seems to be the right level for optimizing query evaluation and storage. At the semantic level, constructs must proliferate until data applications can be modeled naturally and in human terms.

8. Bibliography

[BN] H. Biller and E. J. Neuhold. Semantics of data bases: The semantics of data models. Information Systems 3:1, 1978, pages 11-30.

[Br] M. L. Brodie. Specification and verification of database semantic integrity. Computer Systems Research Group Report CSRG-91, University of Toronto, April 1978.

[Ca] R. G. G. Cattrell. Design and implementation of a relationship-entity-domain data model. Xerox report CSL-83-4, May 1983.

[Ch+] A. Chan, U. Dayal, S. Fox, N. Goodman, D. R. Ries, D. Skeen. Overview of an Ada compatible distributed database manager. Proceedings 1983 ACM-SIGMOD Conference, May 1983, pages 228-237.

[Ch] P. P-S. Chen. The entity-relationship model--toward a unified view of data. ACM Transaction on Database Systems 1:1, March 1976, pages 14-32.

[Co] E. F. Codd. Extending the database relational model to capture more meaning. ACM Transactions on Database Systems 4:4, Dec. 1979, pages 397-434.

[EEMIS] The conceptual model for EEMIS (Energy Emergency Management Information System) and implementation considerations. A. S. Kydes and D. Maier, editors. Prepared by the Laboratory for Software Systems Research, Department of Computer Science, SUNY at Stony Brook, under subcontract to Department of Energy and Environment, Brookhaven National Laboratories, October 1979.

[Ha] L. R. Harris. ROBOT: A high performance natural language interface for database query. Dartmouth College, Department of Mathematics technical report 77-1, February 1977.

[HM] M. Hammer and D. McLeod. Database description with SDM: A semantic data model. ACM Transactions on Database Systems 6:3, Sept. 1981, pages 351-386.

[M+] D. Maier, D. Rozenshtein, S. C. Salveter, J. Stein, D. S. Warren. Towards logical data independence: A relational query language without relations. Proceedings 1982 ACM-SIGMOD Conference, June 1982, pages 51-60.

[MRS] D. Maier, D. Rozenshtein, J. Stein. Representing role in universal scheme interfaces. Oregon Graduate Center report CS/E 83-01, March 1983.

[MS] D. Maier, S. Salveter. Supporting natural language update in database systems. Proceedings European Conf. on AI, July 1982, pages 244-249.

[MW] D. Maier, D. S. Warren. Specifying connections for a universal relation scheme database. Proceeding 1983 ACM-SIGMOD Conference, June 1982, pages 1-7.

[Ro] D. Rozenshtein. Query and Role Playing in the Association-Object Data Model. Ph.D. Thesis, Dept. of Computer Science, State University of New York at Stony Brook, June 1983.

[SM] S. C. Salveter, D. Maier. Natural language database update. Proceedings of the 20th ACL Meeting, June 1982, pages 67-73.

[St+] M. Stonebraker, E. Wong, P. Kreps and G. Held. The design and implementation of INGRES. ACM Transactions on Database Systems 1:3, September 1976, pages 189-222.

[We] J. A. Weeldreyer. Structural aspects of the entity-category-relationship model of data. Computer Science report HR80-251:17-38, Honeywell Corporate Sciences Center, March 1980.

[Zl] M. M. Zloff. Query-by-example: a data base language. IBM Systems Journal 16:4, 1977, pages 324-343.