Database Support for Programming Environments

Michael L. Powell

Mark A. Linton

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

A significant amount of programmer activity in understanding, changing, and debugging software is information management. To address this need, the OMEGA programming environment uses a relational database system to manage program information, and a menu-driven, graphics-based user interface to view, access and update software.

In this paper, we show how to represent programs in the relational model and how traditional programming tasks such as symbol table management are simplified through use of a database. We suggest extensions to relational systems to support more efficient access of recursive data structures, queries involving transitive closure and allow application-specific detection of triggers. These extensions can be used for applications other than programming environments.

1. Introduction

Software constantly changes as new features are added, bugs are fixed, and new hardware technology is exploited. Programs are rarely self-contained entities; they use and interact with algorithms, data structures and subroutines from existing programs or libraries. Consequently, programmers need to understand existing software in order to use and modify it to meet new requirements as well as to create additional, compatible, software.

Most programs are too large to understand in complete detail; hence, programmers select different views to understand different aspects. Understanding the implementation of a procedure may involve looking at its statements according to the program structure; understanding how a variable is manipulated may involve looking at the statements that access the variable; understanding how a running program reaches a certain state may involve looking at statements in the order in which they are executed. To support these and many other views of the same program, we are designing a system, called OMEGA, that stores the procedures, statements, variables, and other information that makes up a program in a database.

Since we wish neither to constrain the range of possible queries nor to duplicate existing facilities, OMEGA uses a general purpose database system. In addition to allowing general queries and multiple views of data, database systems manage permanent storage, support efficient data access, provide concurrency control, attempt to recover from crashes, and try to ensure the integrity of the data. All of these problems arise in software development systems; the fact that database researchers are solving them allows us to address issues specific to programming environments.

We are investigating our ideas by using the relational system INGRES [Stonebraker, Wong and Kreps 76] to manage program information. The relational model was chosen because of the power it offers in expressing queries and describing views. Flexibility in describing views of programs is more important than having a close resemblance to the traditional representation of programs such as that provided by the network data model.

We have implemented a parser that reads statements in a typical programming language and stores program structures in an INGRES database. We have built a prototype interactive interface for viewing and modifying programs in the database.

In the remainder of this paper we describe the OMEGA approach for storing program information in INGRES, present some examples of how the information can be used, and suggest some extensions to relational systems that would increase the power and improve the efficiency with which program information could be accessed. In addition, we show how the use of a database supports debugging facilities by providing relational views of program execution.

2. Storing Procedures, Statements, and Expressions

The traditional format of program source is *text* – ordered, variable-length lines of characters. Because text is organized linearly, it is expensive to extract program semantics from and, therefore, inefficient to use in processing most queries. Simply distinguishing comments from program statements requires scanning each character. Non-trivial queries, such as finding all the uses of the "+" operator in which both operands are integers, requires parsing and semantic analysis of the entire program.

Research supported by the National Science Foundation grant MCS-8010686, the State of California MICRO program, and the Defense Advance Research Projects Agency (DoD) Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

```
prevmax := max;
if a > b then
   max := a;
else
   max := b;
end if;
```

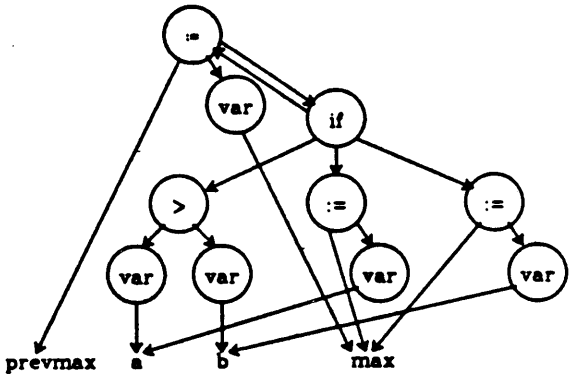

Figure 1: Sample program fragment.

Text provides only weak support for program modifications. Not only does it frequently require redundant typing (e.g., longer, descriptive variable names must be typed each time they are referenced), but it also does not allow higher level operations (e.g., re-order the parameters to a procedure). Operations on program objects (statements, expressions, and variables) require operations on text objects (lines, words, and characters) that only indirectly and imperfectly correspond to them.

As an output medium, text is acceptable only with elaborate conventions for two-dimensional arrangement. Programmers do not read programs as people read novels; both the micro and macro structure of the program should be apparent from its visual presentation. Although it is possible with discipline to arrange text in a form that is understandable, failure to be consistent (e.g., by mis-indenting a list of statements) can mislead the programmer without warning.

To process queries on program semantics, information similar to that which a compiler builds during its parsing and semantic analysis phases is necessary. This consists of some form of program graph and symbol table. Figure 1 shows an example fragment of a program graph for an assignment statement followed by an **if** statement.

This fragment is a list of statements that, ideally, we would store in an *ordered* relation [Stonebraker et al. 82]. Until we are able to use them in INGRES†, we use explicit links; each statement contains a reference to the preceding and following statements. The tables in figure 2 show how the information in the graph in Figure 1 could be stored in INGRES. To keep the example small, we have limited ourselves to program statements, expressions, and variables. In particular, the type information that would be associated with variables and expressions has been omitted.

statements relation

id	class	value	prev	next
1	asgstmt	2	0	3
3	ifstmt	4	1	0
5	asgstmt	6	0	0
7	asgstmt	8	0	0

asgstmts relation

id	var	expr
2	9	10
6	11	12
8	11	13

ifstmts relation

id	cond	then	else
4	14	5	7

expressions relation

id	operator	left	right
10	variable	11	0
12	variable	17	0
13	variable	18	0
14	>	15	16
15	variable	17	0
16	variable	18	0

variables relation

id	name
9	prevmax
11	max
17	a
18	b

Figure 2: Representation of Figure 1 program fragment in INGRES.

†As of this writing, the INGRES implementation of ordered relations is not yet available for general use.

Each tuple in the *statements* relation corresponds to a program statement. We associate an identification (ID) number with each tuple and use the number to refer to that tuple from other tuples. Since some statements can contain an arbitrary number of other statements, this unique key is required to associate all of the contained statements with the containing statement. Statements may be nested in other statements to arbitrary depth. The ID numbers thus also provide a way to represent a hierarchy in a relational database.

Many program objects are like statements in that they may contain objects of their own kind. We call data structures to represent such objects *recursive data structures*. ID numbers represent instances of recursive data structures from within other structures.

OMEGA allocates ID numbers and can request a tuple from INGRES using its relation and ID number. Unfortunately, this interface does not allow INGRES to retrieve the data efficiently nor does it allow OMEGA to perform the queries it needs.

Consider, for example, the relations introduced earlier. If we wanted to print an *ifstmt* tuple, we might use the following algorithm:

```
printstring("if ");
printexpression(if-condition);
printstring(" then");
new_line; indent(+4);
printstatement(if-then-part);
indent(-4); new_line;
printstring("else");
new_line; indent(+4);
printstatement(if-else-part);
indent(-4); new_line;
printstring("end if;");
```

A straightforward implementation of this code generates separate queries for the *if-condition* expression, and the *if-then-part* and *if-else-part* statement lists. Performing several independent queries is more expensive than a single, larger query because the database system can optimize operations for the larger query. We therefore use an alternate approach.

An attribute has been added to each of the *statements* and *expressions* relations to indicate in which procedure they are located. Before processing any part of a procedure, one query is used to retrieve all the *statements* and *expressions* tuples into memory. The individual queries are then performed on this in-memory data. Although this provides more efficient access, the mechanism is outside the normal database system, and thus only a short-term solution.

There are some problems raised by this scheme which must be understood before suggesting database extensions to replace it. Retrieving all the information at once for a large procedure is undesirable because the user must wait for the entire procedure to be retrieved before viewing any part of it. This wait would be particularly annoying if only a simple query needed to be done. For example, to display the statements that reference a particular variable it is not appropriate to retrieve all the statements in all the procedures containing references to the variable. A second problem is that, since the database is not aware of the semantics of the IDs, it will be difficult for it to know which tuples are best cached in memory. The standard cache consistency problems must also be addressed.

A second issue is raised by the recursive nature of program structures. Consider the query that asks for all the statements that reference a particular variable. This query needs to examine the expressions in a statement and all subexpressions of those expressions, to whatever depth expressions are nested in the statement, to discover whether or not the variable is in the statement. There is presently no way to express queries that involve a transitive closure, such queries can only be made by performing many smaller queries.

3. Managing Recursive Data Structures

The issues of efficient access to, and transitive closure queries on, recursive data structures can be solved only by having the database system understand the recursive nature of the data. We propose to supplement the standard database value domains of integers, strings, etc., with a domain of *tuple references*. This would provide information the database system could use to pre-fetch or retain tuples likely to be accessed. In addition, the transitive closure of a tuple reference can be defined and used in queries.

Tuple references are similar to unique ids as proposed in [Codd 79], and closely resemble the data type *tupleRef* available in the Cedar database system [Brown, Cattell, and Suzuki 81]. We have extended these ideas, allowing tuple references to be manipulated through the query language; such usage may implicitly cause the retrieval of tuples. We now examine this proposal in more detail.

3.1 Tuple References

A tuple reference denotes a tuple in some relation in the database. We use the notation "A = **ref**" to define the attribute A as a reference to a tuple. The only difference between tuple references and other fields of relations is that *their values are generated and interpreted by the database system*. All normal database operations apply to tuple references. Additional operations, described below, are also valid.

Although there are several possible implementations of tuple references, we assume that it is always possible to determine in which relation a referenced tuple is by saying *relation*(r), where r is the tuple reference. Thus, without loss of generality, we may think of a tuple reference as a pair (relation, tuple identifier), even if the implementation is otherwise. The value of a tuple reference is generated automatically and is independent of the physical location of the tuple. One distinguished value that any tuple reference can have is a reference to no tuple, similar to the value *nil* in many programming languages.

Often an attribute always refers to a particular relation; in this case we use the notation "A = **ref** R", where R is the name of the relation. This improves the readability of attribute definitions and also allows the database

system to perform optimizations such as minimizing the space needed to store a tuple reference.

Whereas the *ifstmt* relation in figure 2 would have been defined as

```
ifstmts (
   id = integer,
   cond = integer,
   then = integer,
   else = integer
),
```

using tuple references it would be defined as

```
ifstmts (
   cond = ref expressions,
   then = ref statements,
   else = ref statements
).
```

The value of a range variable in a query is the tuple reference for a tuple in the associated relation. For instance, the following example creates an *ifstmt* (which requires a condition expression, then statement, and else statement).

```
range of c is expressions
range of t is statements
range of e is statements
append to ifstmts(cond=c, then=t, else=e) where
   {predicates to select the c, t, and e we want}
```

In addition to normal database operations, it is possible to *dereference* a tuple reference by qualifying it with an attribute name. For example, the following query finds the if statements that have a condition that is simply a boolean variable:

```
range of i is ifstmts
retrieve (i.all) where
   i.cond.operator = "variable"
```

If the specified attribute of a tuple reference is itself a tuple reference, it too may be dereferenced. It is therefore possible to qualify "i.cond" as a normal range variable (in this case, of the expression relation), and refer to its *operator* attribute as "i.cond.operator".

Performing a dereference requires the database system to retrieve the referenced tuple. Indiscriminate dereferencing can cause performance problems similar to the use of IDs that the database system does not understand. However, enough information is available for the database system to apply optimization and caching techniques to improve performance.

3.2 Multi-relation Tuple References

It is often advantageous to have an attribute that can refer to one of several relations. For example, a tuple in the *statements* relation contains a reference to a tuple in one of the individual statement relations, such as *ifstmts* or *asgstmts*. Although storing references to different relations presents no problem to the database system, it is necessary to provide a means to determine the relation that contains a tuple designated by a tuple reference. This facility is provided by the *relation* operator. For example, to find all the if statements we would say

```
range of s is statements
retrieve (s.all) where
   relation(s.value) = "ifstmts"
```

3.3 Transitive Closure Queries

Some properties of programs are transitive. For example, if a variable is used in an expression on the right-hand side of an assignment statement, then it is also used in the assignment statement. Suppose we define the following relation:

```
uses(user=ref, thing=ref)
```

When we define the expression that contains v, a reference to a variable, we add the tuple (e, v) to the *uses* relation where e refers to the expression. When an assignment statement is created with e as the right-hand side, we add the tuple (s, e), where s refers to the statement.

To determine if the variable x is referenced in some statement y, it is necessary to ask if there exist tuples in *uses* (y, a_1), (a_1, a_2), ..., and (a_N, x) for some sequence of $a_1, \ldots, a_N$, $N \geq 0$. This question is simply a matter of determining if (y, x) is in the transitive closure of the relation *uses*.

We define "closure(R)" to be the relation that represents the transitive closure of a binary relation R. The statements that use the variable named "a" can then be found by saying

```
range of s is statements
range of e is expressions
range of v is variables
range of u is uses
range of uclosed is closure(uses)
retrieve (s.all) where
   u.user = s and u.thing = e and
   uclosed.user = e and uclosed.thing = v and
   v.name = "a"
```

4. Execution Information

Most of a programmer's activity in debugging a program consists of trying to answer questions about the program's execution. Traditional debugging tools allow users to ask questions during execution such as "Where am I?", "What is the value of variable a?", and "Where does the value of a change?". They cannot, however, easily handle more complicated questions such as "From the current point in execution where could procedure p next be called?" or "When is procedure p called from procedure q?".

We view debugging as performing queries and updates on a database that contains program execution state information as well as source code information.

This model allows debugging facilities to be easily integrated into the programming environment, since the same user and database interfaces can be used during debugging that are used for program construction.

The idea of having a unified model of program and data is not new; programming systems such as Interlisp [Teitelman and Masinter 81] offer such a view. However, there are two important differences in our approach. We are using a much more powerful data structure, relations, as opposed to lists. Furthermore, we do not use an interpreter, which can be expensive, but rather, execute programs compiled into machine instructions.

Although programs will run directly on the hardware, the user is able to view runtime information as though it were in the database. The program monitor makes this possible by providing the interface between the database system and the executing program shown in figure 3.

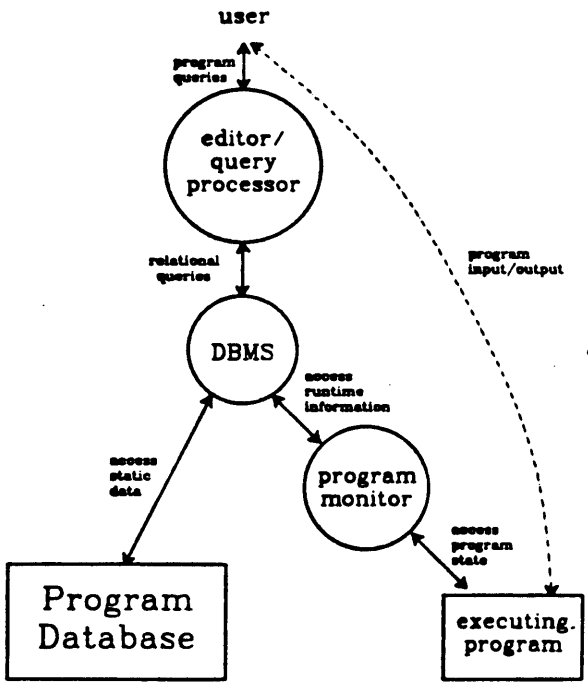

Figure 3: Program Monitor Interface.

The user interacts with the programming interface to construct and view programs stored in the database. This interface translates directives expressed in programming terminology into database commands and displays the results in a traditional program form. In response to a query, the database system may interrogate its own storage facilities or it may request data from the program monitor. The user can also interact with the executing program through the standard input and output interfaces (e.g., a terminal).

The following examples demonstrate the power this approach offers for processing debugging commands. Suppose the programmer wishes to have the program stop whenever the procedure named "buggy" is called. To do this, we use a trigger [Eswaran 76] that might be expressed as

```
range of p is procedures
when iscalled(p) where p.procedure.name = "buggy"
  stop-program
```

Both the trigger condition function *iscalled* and the procedure *stop-program* are implemented by the program monitor. The exact implementation will depend on the hardware and operating system facilities, but all systems provide some way for setting breakpoints in programs. The speed of program execution when triggers are active will depend on judicious selection of breakpoints and efficient evaluation of trigger conditions.

Now suppose that the programmer wishes to have the name and return value of each integer-valued function printed when it returns. This could be requested by saying

```
range of p is procedures
when returns(p) where p.returntype = "integer"
  print(p.name, p.returnvalue)
```

Most debugging facilities can handle the first type of request, stopping when a particular procedure is called; few can handle the second. The availability of source code information in the database makes it possible to easily determine which procedures are integer-valued functions. Access to runtime information makes it possible to retrieve a runtime value in the same way as a normal database value.

To understand the interaction between the database system and the program monitor, we can think of the program monitor as defining a relation

```
runtime(object = ref, value = Value)
```

where *Value* is an INGRES abstract data type (ADT) [Stonebraker 82]. The program monitor implements the operations allowed on values. The *runtime* relation therefore can be used to make a view (such as *procedures* above) containing both static, compile-time data and dynamic, run-time data.

Using triggers in conjunction with access to runtime information provides an extremely powerful mechanism for viewing the execution of a program. Conventional debuggers provide a limited set of events and conditions that may be brought to the attention of the user. Often this means the programmer has the poor choice of too little data or too much. The OMEGA debugger provides a general way for the user to select those events and that information that is most useful. Moreover, output provided by the debugger can be immediately entered into the database. This provides a powerful mechanism for obtaining and examining execution traces.

To provide this facility, we require database extensions related to trigger processing. General triggers are difficult to implement efficiently, but debugging events can often be detected easily. For example, a call to a particular procedure can be trapped without degrading performance by temporarily putting an illegal instruction at the beginning of the procedure.

The ADT facility of INGRES allows the program monitor to implement the operations defined on runtime data. In addition, the database system must provide a way for the program monitor to indicate that a particular trigger condition is true. This would allow the power of the database and triggers to be extended to other kinds of data.

5. Display Interface

Displaying program information on a terminal is a matter of translating the internal program tree and symbol information into a human-readable, perhaps graphical, form. Unlike a browser such as TIMBER [Stonebraker and Kalash 82], the text representing a program does not correspond in any simple way to the tuples and relations that are used to generate it. In particular, there is no way to calculate the number of tuples needed to fill up the terminal screen.

For two reasons, the current implementation of OMEGA manages a data structure outside of the database that contains the displayed form of the program. The first reason will continue to be true: although we can always redisplay any part of the program, doing so is expensive; so we keep previously formatted information around. We hope the second reason is temporary: the database system currently does not allow us to define a set of information to be a "formatted view" of other information. What is required is a way to define a set of tuples that are "under surveillance" and locked, and a way to cause recomputation of some data when specified tuples are changed.

Our present implementation is inadequate for two reasons. First, we must implement database browsing operations such as forward and backward scrolling. Second we must keep the database and displayed views of the data consistent without the benefit of database transactions.

Ordered relations and the *portal* mechanism proposed in [Stonebraker and Rowe 82], should provide what we need. We could store the displayed form of the program in an ordered relation and use portals for browsing. Portals support scrolling and provide locking of the portion of the relation being displayed. Since both the internal and external representations are in the database, transactions must update both representations to ensure that the data is kept consistent.

It would be expensive to have the correct displayable form of the entire program always stored in the database. Some changes to the database could alter a significant amount of the displayable form, yet most portions will not actually be displayed before they are changed again. Therefore, as a user browses through a program, OMEGA will compute the displayed form of any parts not recently displayed.

6. Symbol Table Management

In a compiler, a symbol table provides a means for finding an object associated with a particular name in a given context. Context-dependent name resolution is an important aspect of a good program development system. People tend to build a "mental working-set" of objects and make frequent references to them, using names that would be ambiguous if the context were ignored. Using the approach of [Rowe 82], this function can be performed on the database by a query on a relation that has a name attribute and an attribute that identifies the context.

As an example, consider name resolution in a block-structured language. The basic unit of naming in such languages is called a *block*, within which names must be unique. The *scope* of a block is a collection of blocks that are searched in some order when resolving a name. Suppose we have the following declarations in a Pascal program:

```
procedure A;
  var C : integer;
  procedure B;
    var C : integer;
  end;
end;
```

There is a block associated with each of the procedures A and B. The scope of A consists of only A; the scope of B is the set {B, A}.

Suppose we have the following relations:

```
symbols (
  name = string,
  block = integer,
  value = ref
)
context (
  block = integer,
  scope = integer,
  level = integer
)
```

For the example, *context* contains the tuples (A, A, 1), (B, B, 2), and (B, A, 1). The *level* attribute determines the precedence of blocks in a scope. In block B, the name "C" refers to the C defined in B since the level of (B, B, 2) is higher than that of (B, A, 1).

If the *context* relation is ordered by level, the symbol with name x in block y can be found by taking the first tuple from the result of the following query:

```
range of s is symbols
range of b is context
retrieve (s.all) where
  b.block = y and
  b.scope = s.block and
  s.name="x"
```

7. Version and Configuration Management

Although software changes over time, it is not always the most up-to-date copy that is of interest. Organizations often must support older releases while

developing new ones. A *version* is a snapshot of a program or part of a program at a particular moment of time. Because of, and despite, greater portability of software, it is often necessary to support different but largely identical pieces of software for different hardware or application environments. A *configuration* is a specialization of a program or part of a program to meet a particular set of constraints. The difference between versions and configurations is that versions are ordered in time, with newer ones presumed to supercede older ones, whereas all configurations are equally important, and may coexist forever.

At the core of both version and configuration management are two requirements that differ from most database applications. The first is that there must be several valid and consistent instances of data in the database. The second is that it must be possible for multiple users to access and update these instances of data concurrently. This is not always concurrent access in the usual database sense; it is sometimes convenient to allow new instances of data to be created that will subsequently be coalesced into a single instance.

When a new version of a program is created, it would be inefficient to duplicate the database. Doing so would also make it more difficult to establish the relationship between the old and new versions. Version control systems such as SCCS [Rochkind 75] use a differential file to compactly store program versions. The original version of the file is kept as are all updates necessary to transform the file to the latest (and all intermediate) versions. Hypothetical relations [Stonebraker and Keller 80] can be implemented using the same technique and can be used to provide version control for programs stored in a database.

One of the problems with systems like SCCS is that they require the user to explicitly state when new versions are created. Hypothetical relations do not solve this problem since there is no way to have old versions automatically removed. To save space and speed up queries involving past versions, the user must explicitly dispose of old versions. Coalescing of versions is also a manual process; the exact semantics of a change to an old version is a complex issue currently being studied.

Configuration management involves automatically building a program out of its various pieces according to the given parameters. To minimize the time it takes to build an executable program, only the pieces that have changed or depend on pieces that have changed should be recompiled.

Tools such as *make* [Feldman 78] provide this service, but require the user to specify the program interdependencies. *Make* uses an auxiliary file that contains dependency information; this file must be continually updated by the user as the program changes. Since *make* uses a text file as its basic unit of software and files usually contain several procedures, it also often recompiles more code than is necessary. Using a database, dependency information is not duplicated and the build process can be done without any user assistance. Moreover, the information is directly retrievable at whatever granularity is desired. For example, to find all the procedures that depend on a procedure named "changed" we could say

```
range of p is procedures
range of s is statements
range of uclosed is closure(uses)
retrieve (p.all) where
   uclosed.user = p and uclosed.thing = s and
   relation(s.value) = "callstmt" and
   s.value.proc.name = "changed"
```

Configuration management also requires the ability to determine which program information belongs to which configurations. A common way to implement this feature in conventional programming systems is with conditional compilation facilities. Simple control statements are introduced to indicate which statements ought to be compiled for different configurations. The database provides more complete control over which program elements relate to which configurations, since potentially each object could be tagged with a set of configurations. Generating a configuration would involve restricting a query to a particular tag value.

The most important idea that databases bring to version and configuration control is that *a version or configuration is a view of the program*. To get the most out of this notion, it is necessary that the difficult problems of view updates and consistency be solved. We are providing some important applications to motivate the search for solutions, and look forward with great expectations for future database systems that can support these kinds of operations.

8. Conclusions

Storing program information in a general purpose database system provides a powerful mechanism for manipulating existing software. We are constructing a programming environment that uses a relational database system to manage all program information. This will enable programmers to more rapidly develop, modify, and debug programs.

Ordered relations and variable length strings have been added to the relational model for text processing; these are also useful for representing program information. Hypothetical relations could be useful in supporting version control. In addition, we have suggested three extensions to improve support for programming environments: tuple references, a transitive closure operator, and user-implemented triggers. Tuple references are similar to other proposals for managing unique ids, but can be dereferenced in a way that implicitly requires the database system to retrieve tuples.

These extensions do not represent a radical change in the relational model and are sufficiently general to be of use to a wide variety of applications. For example, computer-aided design (CAD) systems for integrated circuits must manage both hierarchical and relational data and could use a construct like tuple references.

Data management is a fundamental problem of computing. For general purpose database systems to be useful through a wide variety of applications, they must provide primitives for data modeling and access. In analyzing the database needs of a software management system, we have tried to identify those features that will provide the most leverage for manipulating complex data structures.

9. References

[Brown, Cattell, Suzuki 81]
Brown, M., Cattell, R., and Suzuki, N., "The Cedar Database System", *Proceedings of the 1981 ACM Conference on the Management of Data*, Ann Arbor, Michigan, May 1981.

[Codd 79]
Codd, E. F., "Extending the Database Relational Model to Capture More Meaning", *ACM Transactions on Database Systems*, Vol. 4, No. 4, December 1979.

[Eswaran 76]
Eswaran, K., "Specifications, Implementations, and Interactions of a Trigger Subsystem in a Integrated Database System", *IBM Research*, RJ 1820, San Jose, Ca., August 1976.

[Feldman 78]
Feldman, S. I., "Make – A Program for Maintaining Computer Programs", Bell Laboratories, Murray Hill, New Jersey, 1978.

[Rochkind 75]
Rochkind, M. J., "The Source Code Control System", *IEEE Transactions on Software Engineering*, Vol. SE-1, December 1975.

[Rowe 82]
Rowe, L., *private communication*.

[Stonebraker 82]
Stonebraker, M., "Application of Artificial Intelligence Techniques to Database Systems", Electronics Research Laboratory, University of California, Berkeley, Ca., Memo 82/31, May 1982.

[Stonebraker et al. 82]
Stonebraker, M., Stettner, H., Kalash, J., Guttman, A., and Lynn, N., "Document Processing in a Relational Data Base System", Electronics Research Laboratory, University of California, Berkeley, Ca., Memo 82/32, May 1982.

[Stonebraker and Kalash 82]
Stonebraker, M., and Kalash, J., "TIMBER: A Sophisticated Relation Browser", Electronics Research Laboratory, University of California, Berkeley, Ca., Memo 82/17, January 1982.

[Stonebraker and Keller 80]
Stonebraker, M., and Keller, K., "Embedding Hypothetical Data Bases and Expert Knowledge in a Data Manager", Proc. 1980 ACM-SIGMOD Conference on Management of Data, Santa Monica, Ca., May 1980.

[Stonebraker and Rowe 82]
Stonebraker, M., and Rowe, L., "Database Portals: A New Application Program Interface", Electronics Research Laboratory, University of California, Berkeley, Ca., Memo 82/80, November 1982.

[Stonebraker, Wong, and Kreps 76]
Stonebraker, M., Wong, E., and Kreps, P., "The Design and Implementation of INGRES", *ACM Transactions on Database Systems*, Vol. 1, No. 3, September 1976.

[Teitelman and Masinter 81]
Teitelman, W., and Masinter, L., "The Interlisp Programming Environment", *Computer*, Vol. 14, No. 4, April 1981.