Check for updates

# RELAX - The Relational Algebra Pocket Calculator Project

**Victor Matos** and **Rebecca Grasser**
Department of Computer and Information Science
College of Business Administration
Cleveland State University
Cleveland, Ohio 44114 USA
<matos@cis.csuohio.edu>, <rgrasser@acm.org>

**Abstract**

Database courses benefit from the abundance of commercial SQL systems available today. Unfortunately, the same cannot be said about the relational algebra query language. This article considers a lab experience to integrate the learning of these two important topics. In the process of implementing the project, the student acquires practical knowledge in areas such as database programming, parsing and compiling, dynamic SQL code generation, object linking and embedding technologies (OLE), and problem solving skills using the framework of relational algebra. This activity is applied to a traditional second semester database theory course and appears to be very beneficial to the student.

Reviewed Papers

## 1. Introduction

In this paper we describe a laboratory experience appropriate for an advanced database systems course. This project, called RELAX (Relational Algebra Explorer), involves the implementation of an interpreter of Codd's [3, 4] relational algebra expressions. The relational algebra operators are customarily used to introduce the concept of data retrieval in a relational database. However, the lack of practical tools may mislead students to believe that this material is only of theoretical interest having little or no value in the actual development of the database professional. Our belief is that the mastery of relational algebra is the foundation needed for the student to effectively craft any query in any of the commercially available database languages.

There are many compelling reasons to emphasize the use of relational algebra as a query language. It is compact, platform-independent, and relatively simple. Querying with relational algebra, forces the student into a disciplined reasoning process which involves a partitioned, piece-meal, sequential scheduling of the tasks. The ability to break down a large problem into smaller, solvable and articulated subproblems is analogous in many ways to the practice of structured programming techniques emphasized in computer science courses.

In practice, the database instructor uses SQL [1, 2] as the primary vehicle for database interaction. SQL is an important skill to be mastered by the student. However, SQL has less sequentiality than relational algebra, and the reasoning process tends to be more loosely coupled. Additionally, the SQL interpreter/optimizer performs more background actions on behalf of the programmer, resulting in less control of the query.

In this lab experience we present a project to tie together the best of each approach. The student learns to reason queries in an algebraic style, while polishing skills in SQL programming. It is our belief that this project makes an excellent capstone assignment exposing students to application of concepts including algorithms and data structures, database and information retrieval, programming languages, and software methodology and engineering.

## 2. RELAX - The Relational Algebra Pocket Calculator

The pocket calculator consists of three components: (1) a database *explorer* module, (2) a syntax directed *translator,* and (3) a *library* of procedures implementing algebra operators and relational database schema modifiers.

The explorer component facilitates the examination of the database dictionary. For simplicity we use the Microsoft Access database model. The metadata describing the database could be used to see the composition of the different tables, fields, queries, relationships, forms, and reports held in the Access database. The translator converts relational algebra expressions requesting data from the Access database into new relations. The mechanism is straightforward; the results of each query are placed into an *answer* table, which could be seen using the database explorer. RELAX supports a total of twelve relational constructs including: aggregation, selection, projection, join, natural join, left outer join, intersection, division, Cartesian product, union, difference, rename, and assignment.

*Example: The Company Database*

Our examples are based on a simplified Company database [5] whose schema is given as follows.

| | |
|---|---|
| EMPLOYEE (fname, minit, lname, ssn, bdate, address, sex, salary, superssn, dno) | KEY: ssn |
| DEPARTMENT (dname, dnumber, mgrssn, mgrstartdate) | KEY: dnumber |
| PROJECT (pname, pnumber, plocation, dnum) | KEY: pnumber |
| WORKS_ON (essn, pno, hours) | KEY: (essn, pno) |
| DEPENDENT (essn, dependent-name, sex, bdate, relationship) | KEY: (essn, dependent-name) |

The Employee table contains personal data about people working for the company. This includes the key value SSN (Social Security Number). The Department table gives the number and name of each department, as well as the manager's SSN number and his/her starting date. The Project table identifies each activity being developed by the company. Works_On describes the employee's weekly workload. The relation Dependent lists the direct family member of each employee.

## 3. RELAX Version of the Algebra Operators

RELAX expressions can be of any degree of complexity. They can be nested one inside of the other using parenthesis. Table 1 summarizes the operators that may appear in a RELAX relational algebra expression. The PRECEDENCE column in Table 1 indicates the relative *importance* of each operator. For instance, Aggregation has a higher precedence (4) than Union (1), therefore the '#' operator should be computed before the '+' operator. More details about processing the precedence grammar are given in the next sections. In general, the calling sequence to invoke the relational algebra parser is *Table*$_1$= *Algebra_Expression,* where *Table*$_1$ is the name of the output relation constructed accordingly to the specifications given in *Algebra_Expression.*

A typical algebraic expression is made according to the following syntax *X(Relation*$_1$ *) OP Y(Relation*$_2$*),* where *X* and *Y* represent optional unary operators and *OP* is an optional binary operator. A description of the valid *X, Y, OP* operators is provided in Table 1 below.

| OPERATOR (Usual notation) | PRECEDENCE | RELAX SYMBOL | DESCRIPTION |
|---|---|---|---|
| Aggregation | 4 | # | Employee # (Dno, Max, Salary) Other functions are: SUM, AVG, MAX, MIN, COUNT |
| Selection | 4 | :() | Employee : ( Sex ="F" and Dno = 5 ) |
| Projection | 4 | [] | Employee [ SSN, Lname, Salary ] |
| Join | 3 | { } | Employee { Ssn = Essn } Dependent |
| Natural Join | 3 | * | Employee * Skills |
| Left-Outer Join | 3 | >> | Employee >>{Ssn = Essn} Dependent |
| Intersection | 2 | & | Engineers & Managers |
| Division | 2 | / | Works_On[Essn,Pno] / Projects[Pnumber] |
| Cartesian | 2 | ** | Projects ** Departments |
| Union | 1 | + | MaleEmp + FemaleEmp |
| Minus | 1 | - | Engineers - Managers |
| Assignment | 0 | = | Result = SomeExpression |

Table 1. Relational algebra operators in RELAX

## 4. The RELAX Library

A collection of Visual Basic functions is used to support the processing of algebra expressions. The routines are grouped into two categories: data manipulation and schema operators. The schema operators consist of the functions *addColumn(), deleteColumn(), renameColumn(),* and *showSchema().* These functions give the user the ability to alter the structure of a table by adding, removing, or renaming one column at a time. Other supporting routines include the *noDuplicates()* function which removes duplicate tuples from relations. The data manipulation routines support each of the basic algebra operators. For instance, the algebraic *union* operation "Answer= T1 + T2" is implemented in a procedure called *RelaxUnion(Answer, T1, T2).* This routine checks whether or not the tables T1 and T2 have compatible schemas. If they do, the tables are combined into a new table called *Answer* and duplicate records are removed. The next section suggests how to implement each operator using Access-SQL syntax.

## 5. Simulating Relational Algebra with MS-Access SQL

The following table indicates how to convert each of the basic algebra operators into the SQL dialect supported by MS-Access. The students must individually implement and test each of the operators as a Visual Basic function. An example of such coding is provided below. It is convenient to group all of the algebraic operators into a single global VB module. The same modularization applies to the implementing of the schema operators.

| Algebra Op/ Symbol | RELAX Syntax | MS-Access SQL Version |
|---|---|---|
| **Union** + | result = r1 + r2 | select * into *temp* from r1;<br>insert into *temp* select * from r2;<br>select distinct * into result *from temp;* |
| **Minus** - | result = r1 - r2 | select * into result from r1<br>where not Exists<br>(select * from r2<br>where r1.field$_1$= r2.field$_1$<br>...<br>and r1.field$_n$ = r2.field$_n$ )<br>Note: Scheme(r1)=Scheme(r2)=field$_1$,...field$_n$ |
| **Intersection** ∩ | result = r1 & r2 | select * into result from r1<br>*where exists*<br>(select * from r2<br>where r1.commonfield$_1$= r2.commonfield$_1$...<br>and r1.commonfield$_n$ = r2.commonfield$_n$) |
| **Selection** σ | result = r1 : (condition) | select * into result<br>from r1<br>where (Condition); |
| **Projection** ∏ | result = r1 [A,B,...,N] | select distinct A,B,..,N into result<br>from r1; |
| **Rename** ρ | result = r1 ? (old, new) | select old as new into result<br>from r1; |
| **Aggregate Formation** ℱ | result = r1 # ( GroupBy, Function, Field) | select *group-by-list, function-list* into result<br>from r1<br>group by *group-by-list;*<br><br>where *function-list* is *function (field)*<br>function: Max, Min, Count, Avg, Sum. |
| **Cartesian** × | result = r1 ^ r2 | select * into result<br>from r1, r2 |
| **Join** | result = r1 {r1.a = r2.b} r2 | select * into result<br>from r1, r2<br>where r1.a = r2.b |
| **Natural Join** * | result = r1 * r2 | select r1.*, r2.* into result<br>from r1, r2<br>where (r1.f$_1$ = r2.f$_1$ ) ...<br>and (r1.f$_n$ = r2.f$_n$ )<br>NOTE: f$_1$,...f$_n$ represent the common fields of r1 and r2 |
| **Left Outer Join** => | result = r1 >> r2 | select e.*, d.* into result<br>from Employee e LEFT join<br>Dependent d ON<br>( e.Ssn = d.Essn )<br>Note: Assume r1 is Employee and r2 is Dependent |
| **Division** ÷ | result = r1 / r2 | Assume schemas: r1(A,B) , and r2(B)<br>select distinct x.A into result<br>from T1 as x<br>where NOT EXISTS (<br>select *<br>from T2 as y<br>where NOT EXISTS (<br>select *<br>from T1 as z<br>where (z.A= x.A) AND (z.B= y.B) ) ) |

Table 2. SQL version of the RELAX Operators in the MS-Access Environment.

Reviewed Papers

*Example. A Relax Function*

The routine below illustrates the implementation of the *aggregation* operator. For instance, the expression

dbStatus= RELAX_Aggregation

("myResult", "Employee", "sum", "salary", "sex")

partitions the *Employee* table according to *sex* values. The *sum* of *salaries* is computed on each group. A new table called *myResult* is generated. This table has two rows holding the gender and the total salary for that gender. The function *RELAX_Aggregation* returns the number of rows in the resulting table. In the case of errors the variable dbStatus receives the value –1.

```
Public Function
    RELAX_Aggregation(destinationTable, sourceTable, _
    theFunction, onCol, groupByColumn) As Integer

On Error GoTo Adios
Dim countResult As Integer
Dim db As Database
Dim rs As Recordset
Dim mySQL As String
  countResult = -1
  Set db = CurrentDB
  mySQL = "select " & groupByColumn & ", " & _
  mySQL = mySQL & theFunction & "( " & onCol & " ) " & _
  mySQL = mySQL & " into " & destinationTable & _
  mySQL = mySQL & " from " & sourceTable & _
  mySQL = mySQL & " group by " & groupByColumn
  db.Execute (mySQL)
  Set rs = db.OpenRecordset(destinationTable)
  countResult = rs.RecordCount
Adios:
  RELAX_Aggregation = countResult
End
```

## 6. The Graphical Interface

RELAX has two main screens. The first shows the *database explorer* and the second displays the *calculator* itself. The explorer screen uses a typical file-finder interface to locate and open the database. Once it is selected, a series of list-boxes holding table-name, query-name, relationships, and reports are filled-up. Clicking on a table-name shows all the fields (type/size) in the table as well as a grid-view of the table. A similar procedure is followed for queries, relationships, and reports. For brevity we omit the view of the explorer's screen.

The second view is the calculator. An image of the screen is provided in the Figure 1. The calculator resembles a common pocket calculator. The top textbox is used to enter the algebraic expression. In the making of the algebra expression the user will need to know the exact spelling of tables, and fields. Those names could be seen in the tabbed control on the lower left portion of the screen. The algebra operators are grouped on the upper left box. The screen is context sensitive, for instance, if the mouse is placed on top of the # (aggregation operator) a quick-help screen appears showing the syntax for the operator.

Once the expression is formed, the user pushes the *Parse* button. For pedagogical reasons an image of the postfix version of the query is shown in a textbox. If the query is syntactically correct its results can be seen by pushing the *Show Table* button. Unwanted tables can be deleted using the *Drop Table* button. The image below shows the calculator processing the expression *test1 = (employee:(sex= 'M')[ssn] + (employee:(sex= 'F'))*. Using the selection and projection operators, this query selects the social security numbers (ssn) of those individuals whose gender is male from the Employee table. A similar query is performed to retrieve the social security numbers (ssn) of all female employees. Using the union operator, the two sub-queries are combined to form the final result. The result screen is also displayed.

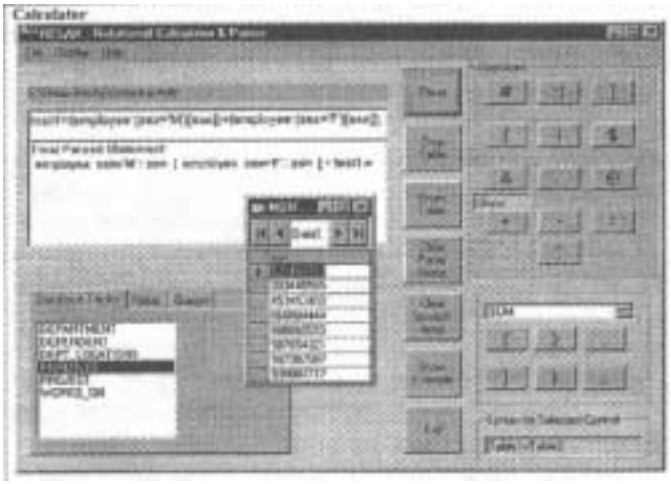


Figure 1. The Graphical Interface

## 7. The RELAX Parser

The job of the parser is to isolate the tokens or components of the input expression, convert the expression from infix to postfix notation, and finally evaluate the query. The code generated by the parser consists of a sequence of calls to functions in the RELAX library. Once the query is evaluated its resulting tuples are stored into a table. The overall strategy is as illustrated with the following example.

*Example: Parsing an Expression*

a. Assume the user supplies the following union query

MyResult = (r1 + r2) [A,B]

The query is broken into tokens. Each token is given to the parser which in turn checks its *type* (operand, operator) and *precedence*.

b. The tokens are rearranged into their postfix order.

| MyResult | r1 | r2 | + | A,B | ] | = |
|---|---|---|---|---|---|---|

c. The code generator finds the + symbol in the postfix string. A call to the *Relax_Union(...)* routine is made. The routine requests the previous two arguments from the postfix string (r1, and r2), and generates the name of a temporary table *_relaxTemp001* in which results for the union will be held. The fragment of code

*call relaxUnion (_relaxTemp001, r1, r2)*

is produced. The postfix expression is changed to

*myResult _relaxTemp001 A,B ] =*

d. The scanning continues, the symbol "]" representing projection is found and more code is generated. At this point a call to the *RELAX_Projection(...)* function is produced. This routine takes the previous two arguments from the postfix list, and replaces them with the new table *_relaxTemp002.* The new postfix string is

*MyResult _relaxTemp2 =*

e. Parsing continues in a similar mode for the rest of the string. The final code is

```
BEGIN
  call relaxUnion
    (_relaxTemp001, r1, r2)                    'Code for UNION
  call relaxProjection
    (_relaxTemp002, _relaxTemp001, 'A,B' )  'Code for PROJECTION
  call relaxAssignment
    ( MyResult, _relaxTemp002)                 'Code for ASSIGMENT
END
```

Finally the query is executed, the table *MyResult* is added to the schema to hold the resulting tuples, and the temporary tables are removed. An interesting experience in this project was the use of system-defined objects and the construction of custom made controls. We asked our subjects to implement the stack services used above by means of their own ActiveX DLL (Dynamic Link Library) control. Details about this subject are out of the scope of this paper.

## 8. Exposing the Database Architecture

RELAX relies on the Microsoft Access 8.0 object model to gather and display information about a particular item [6]. In the MS-Access organization this data is held in several objects whose hierarchical disposition is displayed in the figure below. That hierarchy structure is known as the DAO (Data Access Object) organization. In the DAO hierarchy the *Containers, TableDefs,* and *QueryDefs* collections act as the primary sources of information. For instance, the user could use the *TableDefs* collection to request facts regarding a particular table, such as the associated fields, their data types, sizes, maximum and minimum value, count of different values, dates of creation, last update, composition of the primary key, updateable flag, validation rules, and record-count. The user could also see the actual data records from the selected table.

Similarly, if the user chooses to see what stored queries are available in the database, the explorer examines entries in the *QueryDefs* collection. Information such as the text of the selected SQL statement, its type, parameters, dates of creation and last update, updateable flag, and formal output list is shown. The user may opt for execution of the query and displaying of the results, as well as alteration of the base tables (in the case of action queries).

Other MS-Access objects such as *reports* and *forms* can be remotely examined by executing an instance of the

```
Dim theDB As Database
Dim theDoc As Document
Dim theCont As Container
Dim theProperty As Property
Set theDB = OpenDatabase("c:\Temp\Nwind.mdb")
Set theCont = theDB.Containers!Tables
Debug.Print "Documents in " & theCont.Name & _
            " container"
For Each theDoc In theCont.Documents
    Debug.Print "  " & theDoc.Name
    Debug.Print "Properties of " & _
            theDoc.Name & " document"
    For Each theProperty In theDoc.Properties
        Debug.Print "  " & theProperty.Name & _
                    " = " & theProperty
    Next theProperty
Next theDoc

theDB.Close
```

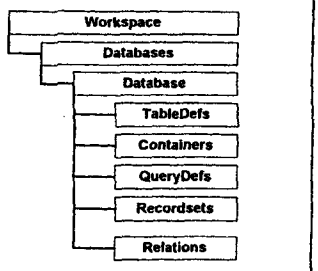



**Figure 2. The MS-Access Database Dictionary**

*Access.Application.* Once the application is running the user's interface changes to the familiar MS-Access menu, from which the report (form) could be browsed or printed. The following fragment of code illustrates the activation of the Access application.

```
Set accessApp = CreateObject("Access.Application")
accessApp.OpenCurrentDatabase (txtTheDatabaseName.Text)
accessApp.DoCmd.OpenReport Report, acViewPreview
accessApp.Application.Visible = True
```

### *Example: Listing Tables from Schema*

The fragment of Figure 2 shows the scanning of the MS-Access *Containers* collection to gather data about documents of type "*Tables*". Each table and their properties is displayed. Similar code could be used to examine queries, reports, relationships, and forms.

## 9. Conclusion

We believe this capstone project provides students the opportunity to integrate several important concepts that are normally learned in various computer science courses such as data structures, compiler design, database systems, and advanced programming. In addition to the chance of reviewing those important topics, the students face a challenging programming project that could very well be developed during a typical semester course. We identified several specific gains for the students as follows.

1. Maturity in *database programming* using Visual Basic language. This goal is achieved by programming at the level of the DAO object control (or an equivalent approach such as ADO). Some of the objects and methods that are studied are *Database, RecordSet, OpenDatabase, OpenRecordSet, Execute.*
2. Exposition to the metadata describing the internal organization of the database, which promotes understanding of the MS-Access architecture and its interfacing with Visual Basic. This portion emphasizes the interaction with the DAO Hierarchy, which includes collections of objects such as *WorkSpaces, TableDefs QueryDefs,* and *Containers.*
3. Ability to formulate and evaluate complex database expressions in relational algebra query language,
4. Ability to formulate sophisticated *retrieval* and *action*

SQL queries.

5. Construction of advanced user-defined data structures using the *Type...End Type* construction,
6. Building of Visual-Basic user-defined objects. Writing of class-modules. Construction of Windows' Dynamic Link Libraries (DLL).

We recognize that areas for improvement in the project include the error detection phase and the handling of database dictionaries of other ODBC-compliant databases such as Oracle and SQL-Server.

## References


[1] American National Standards Institute. Database Language SQL. Document ANSI X3.135-1992.

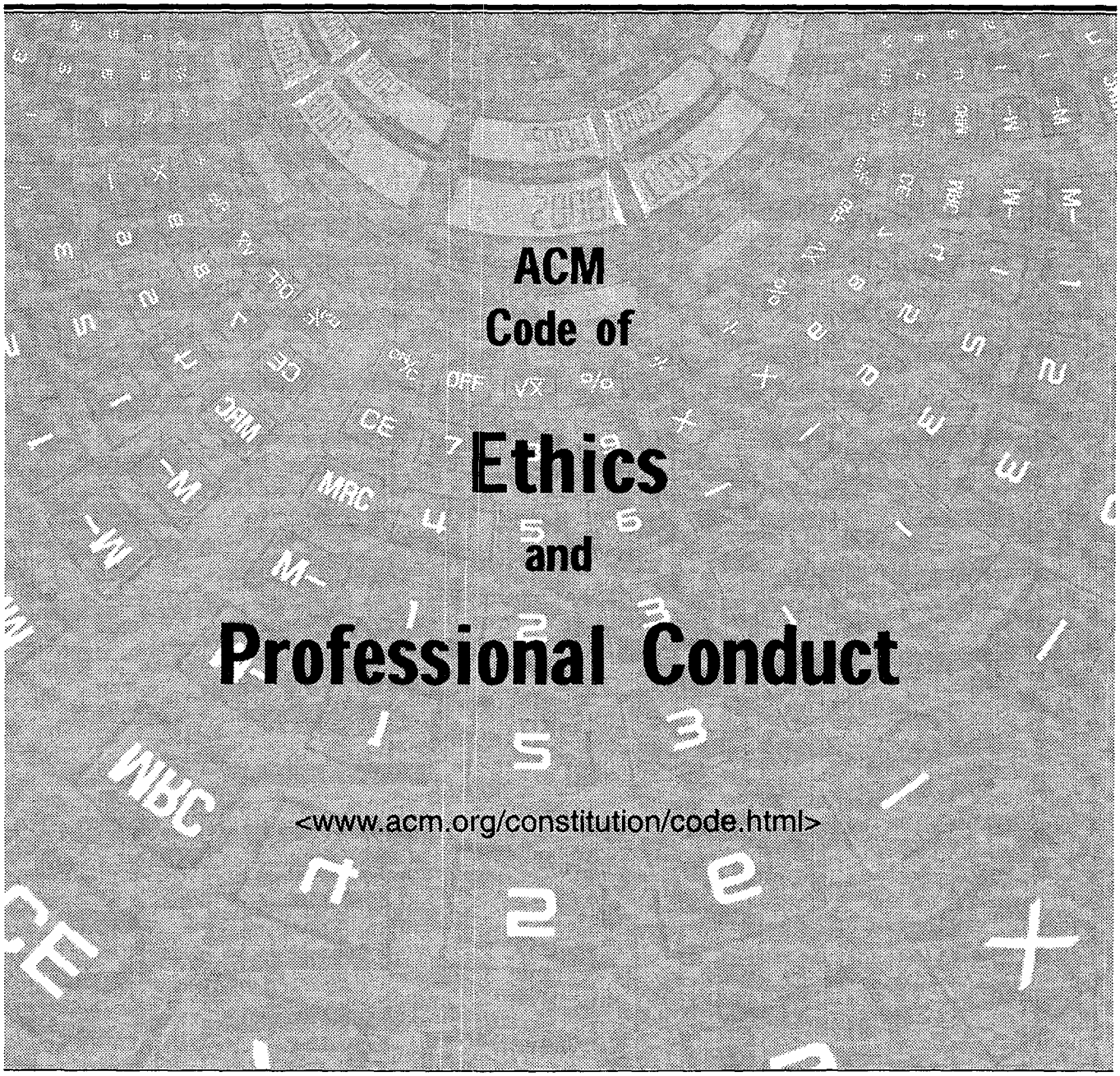
2] Chamberlin, D., et al. "SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control". *IBM Journal of Research and Development*, 20:6, November 1976.

[3] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks". *CACM* 13, No. 6, June 1970.

[4] Codd, E.F., "Relational Completeness of Data Base Sublanguages", In *Database Systems, Courant Computer Science Symposia Series 6*. Englewoods Cliffs, NJ, Prentice Hall, 1972

[5] Elmasri, R., Navathe, SR. *Fundamentals of Database Systems*, Third Edition. Addison-Wesley Publishing Co. 1999.

[6] Microsoft Press. *Microsoft Visual Basic 6.0 Programmers Guide*. 1998.


Reviewed Papers