CHECK DIGITS: DOMAIN IMPLEMENTATION IN MICROCOMPUTER DATABASES

Richard Hartley
James Scott
Roger Hayen
Central Michigan University
College of Business
Office and Information Systems
Grawn 320B
Mt. Pleasant, MI 48859

ABSTRACT: Students studying their first systems learn that many businesses use serial numbers to identify such entities as customers, employees, and products. Sequentially numbered identifiers are prone to substitution and transposition errors during data entry. The addition of a check digit allows a business to continue to use their serial numbers (with modification) and provides identifiers less prone to undetectable data entry errors.

Project oriented systems courses often utilize 4GL tools to implement student designs. With such tools students can reduce the development time considerably over the time required using 3GL procedural programming languages, this allows students to focus on design considerations rather than procedural programming techniques. However, in real life as well as in student projects, some functions cannot be implemented without the use of a procedural language. This article provides instructors material for illustrating how check digits can be incorporated into a microcomputer database application using the procedural language components of two popular database systems.

KEYWORDS: Data Validation, Programming, 4GL, 3GL, Check digits, Self-Checking Numbers.

INTRODUCTION

Within our advanced systems analysis and design course students visit actual businesses where they document a current application, and design a new application. As part of the system design students learn to define a domain of values for each attribute following guidelines taught in previous classes. (1,2,3,4) For example in an accounts receivable application the attribute CUST_ID in the entity PAYMENT would normally have as its domain all the CUST_ID's in the table CUSTOMERS. A validity test is then constructed by checking to see if the CUST_ID in a particular PAYMENT record exists in the CUSTOMER table.

Many of the companies that the students contact create their own identification numbers for entities such as customers, products, and employees, using serial numbers. As students develop systems they realize that such identification numbers are difficult to validate because substitution and transposition errors frequently yield other valid identification numbers. (1,5) If customer number 131 is entered as 113 it may still be a valid number but possibly incorrect number as the number 113 may be assigned to another customer. In a relational database this error leads to problems with referential integrity as records in different tables for the same customer could contain incorrect customer identifiers. The use of self-checking numbers is one solution. However the implementation is not as clear. The student projects are developed using microcomputer database packages such as R:BASE, dBASE, and PARADOX. The students are encouraged to develop as much of each application as is possible using only the 4GL component of the database. They are discouraged from using the procedural language component because we want them to focus on the effective use of 4GL tools in this course. By the students

taxing the 4GL tools to the limit, they learn both the strengths and weaknesses of this approach to applications development.

SELF-CHECKING NUMBERS

A self-checking number is one which contains digits that may be utilized to determine the validity of the number. Credit card numbers, bank account numbers, and the ISBN numbers used in library systems all utilize check digits. The incorporation of self-checking numbers into microcomputer databases can provide the basis for discussions of entity identifiers, data validation, data types, the need for a procedural language component in a microcomputer database, and the use of entry/exit procedures in a 4GL form generator. (6,7)

Check digits are used widely on mainframe systems where the algorithm for computing the check digit is added to procedural program code and maintained in application libraries. Several methods of creating check digits have been discussed in the literature. (8,9) The modulus eleven methodology will be used to illustrate the implementation of self-checking numbers in microcomputer systems.

To calculate a check digit, each position of the identifier is multiplied by a weight. We use arithmetic weights beginning with two for the units position and increasing by one for each succeeding position. The modulus of the sums is then subtracted from eleven and the result is the check digit(s). Naturally, mod 11 will yield the values zero through ten. The analyst must decide at the outset what should be done with the numbers that yield a two digit check digit. The options are to 1) use two digit positions for the check digit (00-10), 2) use a single digit position but use a letter for the number 10, or 3) do not use any identifiers that yield a check digit value of 10.

GENERATING CHECK DIGITS

The implementation of a check digit algorithm is specific to each relational database product. Tables 1 and 2 are examples of the procedural code for dBASE and R:BASE. (10,11) Assuming that record

Table 1: Procedural Code for dBASE IV

```
**********************************************************************
*       Enter a five digit identifier.
**********************************************************************
INPUT "Enter a five digit number:  " CUST_ID
**********************************************************************
*       Separate the digits.
**********************************************************************
NUM_CUST=INT(CUST_ID)
DIGIT1=INT(NUM_CUST/10000)
DIGIT2=MOD(INT(NUM_CUST/1000),10)
DIGIT3=MOD(INT(NUM_CUST/100),10)
DIGIT4=MOD(INT(NUM_CUST/10),10)
DIGIT5=MOD(NUM_CUST,10)
**********************************************************************
*       Determine the check digits.
**********************************************************************
CHKDGT=MOD(6*DIGIT1+5*DIGIT2+4*DIGIT3+3*DIGIT4+2*DIGIT5,11)
**********************************************************************
*       Append the check digits to the original identification number.
**********************************************************************
IF CHKDGT = 10 THEN
        CUST_ID = NUM_CUST + "-X"
ELSE
        CUST_ID = NUM_CUST + "-" + CHR(CHKDGT)
ENDIF
```

Table 2: Procedural Code for R:BASE 3.1

```
*(                                      )
*(Accept a five digit text identifier                  )
*(                                      )
FILLIN CUST_ID=5 USING "Enter a customer identifier:  "
*(                                      )
*(Parse the customer identifier into 5 fields and convert each  )
*( to an integer value.                          )
*(                                      )
SET VAR DIGIT1 TO (SGET(.CUST_ID,1,1))
SET VAR DIGIT2 TO (SGET(.CUST_ID,1,2))
SET VAR DIGIT3 TO (SGET(.CUST_ID,1,3))
SET VAR DIGIT4 TO (SGET(.CUST_ID,1,4))
SET VAR DIGIT5 TO (SGET(.CUST_ID,1,5))
SET VAR DIGIT1 INT DIGIT2 INT DIGIT3 INT DIGIT4 INT DIGIT5 INT
*(                                      )
*(Compute the check digit.                        )
*(                                      )
SET VAR CHKDGT TO
        (MOD(6*DIGIT1+5*DIGIT2+4*DIGIT3+3*DIGIT4+2*DIGIT5,11))
*(                                      )
*(Append the appropriate check digit to the original number.   )
*(                                      )
IF CHKDGT = 10 THEN
        SET VAR CUST_ID TO (CUST_ID + "-X")
ELSE
        SET VAR CUST_ID TO (CUST_ID + "-" + CTXT(CHKDGT))
ENDIF
```

identifiers are maintained as character data, the first step is to break the identifier into as many integer elements as there are digits in the identifier. The second step is the calculation of the check digit(s). The last step is combines the original identifier and the computed check digit (or an "X" if the check digit is "10").

INITIAL ASSIGNMENT OF A CHECK DIGIT

Identification numbers may be assigned independent of the data entry process or as part of data entry. Using the algorithms as they are written, the user would determine the check digit for a single requested identifying serial number. (The link to the procedural code is accomplished by the DO statement of dBASE or the RUN statement of R:BASE in the application menu.) The user would then add new customers, products, etc. using the identifier with the check digit.

Performing the computation of the check digit for each serial number as needed is cumbersome. A more workable approach would be for the user to assign identifiers from a list of valid but as yet unassigned identifiers. Such a list could be produced with minor modifications to the above code. The addition of a WHILE loop would produce a page of numbers from which to make customer identifier assignments for new customers. The user could be prompted for a starting number and the quantity of identifiers desired. (Ideally, the starting number would be determined from the data already in the database.) As a number is used from the sheet, the user would draw a line through the number indicating that the number had been assigned.

10200-3	10205-2
10201-5	10206-4
10202-7	10207-6
10203-9	10208-8
10204-0	10209-X

SYSTEM COMPUTATION OF THE CHECK DIGIT DURING DATA ENTRY

If identifying numbers are to be assigned by the system then the computation of the check digit would be done during data entry. The developer has two choices in developing data entry screens: write a procedural language program or use a 4GL form generator. With the procedural language approach, the developer writes a routine that determines the highest identifier already used, increments the serial number portion of the identifier by one, and computes the check digit for that number.

Some microcomputer databases now support a data type that computes the next available serial number ..., however none can currently generate check digits.

When a 4GL form generator is used, the developer has limited control over the data during data entry. (12,13) Some microcomputer databases now support a data type that computes the next available serial number (called AUTONUMBER in R:BASE), however none can currently generate check digits.

Products such as R:BASE offer entry/exit procedure capability (EEP) for each field of a form. (13) EEP's provide the capability of executing procedural statements before accepting data into a field and after the user has exited the field. If the 4GL supports EEP's then an entry procedure can be defined to perform the same algorithm as illustrated in the procedural language approach.

METHODS OF VERIFYING DATA

If all identifiers are entered with an attached check digit, then the table containing the primary entity (such as CUSTOMERS) will constitute an accurate domain of identifier values. As data for another entity are entered, such as PAYMENTS, the CUSTOMERS table may be used to test the validity of CUST_ID. Substitution and transposition errors will not generally yield another valid CUST_ID as would frequently happen if the check digit had not been utilized. In a procedural programming approach the identifying number might be verified at the time of entry by recalculation of the check digit followed by a comparison with the check digit entered.

IDENTIFIERS CONTAINING ALPHABETIC CHARACTERS

Users may have chosen to utilize abbreviations of customer names for identifiers or actual model numbers of products. How can a check digit be computed for such identifiers?

The process begins by converting each letter to its ASCII equivalent numeric value so that it can be multiplied by a weight. This can be done by using the character-to-ASCII function of your database product, (ichar(string) in R:BASE, val(string) in dBASE). This will yield the numbers 65 to 90 for the letters A to Z and the numbers 48 to 57 for the digits 0 to 9. (The ASCII code table can be found in the manuals for your particular database product.) By subtracting 55 from the ASCII value of each letter and by subtracting 48 from the ASCII value of each number, we arrive at a series of numbers from 1 to 36. Each of these values are multiplied by their respective weight and summed. The final step is to determine the check digit using modulus 37. (Thirty-seven is the closest prime number greater than the number of values represented.) This exercise can be used to further discuss the ASCII code table by asking students how a check digit could be computed if the identifier contained lower case letters or special characters.

RECOMMENDATIONS

1. Use self-checking numbers wherever a serial number is used for identification of an entity. Without the check digit, serial number identifiers are prone to errors because of the substitution or transposition of digits.
2. While 4GL's make rapid system development possible, students need to be aware that a procedural language component is frequently necessary to handle functions not provided by the database vendor.

Future 4GL's will utilize EEP's to provide an easier integration of procedural language code into 4GL forms generation products.

3. A new data type for specifying the generation of serial numbers with check digits would be beneficial to application developers using 4GL tools.

CONCLUSIONS

Check digits provide an instructor with three opportunities. First, they provide an opportunity to discuss the problems associated with using serial numbers as keys. Second, they give a real-world application for using procedural language within a microcomputer database system. Third, they demonstrate a method for implementing domains. This article provides a method for instructing students in the use of check digits and algorithms for their implementation.

REFERENCES

1. Stamper, David, Database Design & Management - An Applied Approach, Mitchell McGraw-Hill, 1990, pp 298, 308-311, 405-406
2. Date, C.J., Relational Database: Selected Writings, Addison-Wesley, 1988, pp. 460-1
3. Codd, E.F., The Relational Model for Database Management, Addison-Wesley, 1990, pp 43-50
4. Codd, E.F., Domains, Keys and Referential Integrity in Relational Databases, InfoDB, May 1988, C.J. White Consulting, San Jose, CA.
5. Whitten, Jeffrey L., Bentley, Lonnie D., and Barlow, Victor M., Systems Analysis & Design Methods, Second Edition, Irwin, 1989, p. 552
6. Fleming, Candace C. and vonHalle, Barbara, Handbook of Relational Database Design, Addison-Wesley Publishing Company, 1989, pp 214-232
7. Tasker, Dan, In Search of Fourth Generation Data, Datamation, July 1, 1987, pp. 61-2
8. Eliason, Alan L., Introduction to Systems Design, McGraw-Hill, 1989, pp 487-9
9. Murray, Thomas J., Computer Based Information Systems, Richard D. Irwin, 1985, pp, 73-7
10. Microrim, R:BASE User's Manual, 1990, third edition, Chapter 2 p. 44, Chapter 6, p. 37
11. Ashton-Tate, dBASE IV Reference Manual, 20101 Hamilton Avenue, Torrance, CA 90502
12. Infoworld, Multiuser Programmable Databases - Part I, November 12, 1990
13. Infoworld, Multiuser Programmable Database - Part II, January 14, 1991, pp 45-60

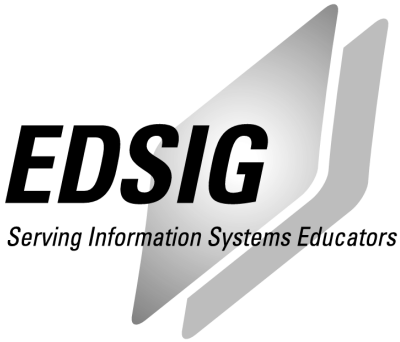
AUTHORS' BIOGRAPHIES

Richard L. Hartley, Ph.D., is an assistant professor of information systems at Central Michigan University. He has done extensive consulting in microcomputer database systems. He teaches system analysis and design, COBOL programming, and database design courses.

James P. Scott, Ph.D., is an associate professor of information systems at Central Michigan University. He teaches database management and beginning systems design.

Roger Hayen, Ph.D., is a professor in the Office and Information Systems Department of the College of Business Administration at Central Michigan University. He has experience as an international consultant in the training for and development of decision support systems.

Information Systems & Computing
Academic Professionals



STATEMENT OF PEER REVIEW INTEGRITY

All papers published in the Journal of Information Systems Education have undergone rigorous peer review. This includes an initial editor screening and double-blind refereeing by three or more expert referees.

ISSN 1055-3096