AN IMPLEMENTATION OF A LOSSLESS JOIN ALGORITHM

by

KARL RICHARD KLOSE

B.S., Bucknell University, 1958
M.S., University of Alabama, 1962
M.A., University of Alabama, 1967
Ph.D., University of Alabama, 1970

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1983

Approved by:

Major Professor

TABLE OF CONTENTS

THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE.

THIS IS AS RECEIVED FROM CUSTOMER.

ACKNOWLEDGEMENTS

The author expresses his sincere appreciation to Professor Elizabeth A. Unger for suggesting this problem and most especially for her patience, guidance and encouragement during its solution.

Chapter 1

INTRODUCTION

Credit for the widespread interest in the Relational Data Model must go to E. F. Codd, who in his landmark paper [Codd,1970] described the application of the elementary theory of mathematical relations to large stores of formatted data. This model represents a significant departure from the hierarchic and network models in both the logical description and manipulation of data.

Simply stated, the relational model views data as a relation or collection of relations. Each relation may be thought of as a two-dimensional table whose columns and rows are termed "attributes" and "tuples" respectively. The order of the columns and rows in a relation is not important. However, it is required that each row in a given relation be distinct and that the elements or "entities" of a given column be members of a well defined "domain" of possible values of like type. A "candidate key" of a relation is any set of attributes of that relation with the time independent properties that each tuple of the relation is uniquely identified by the current values of this set of attributes and that this unique identification property is lost if any attribute is removed from the set. Any

attribute that is contained in a candidate key is termed a "prime attribute" while those remaining are termed "non-prime attributes". Finally, a crucial feature of this model is that associations or relationships among relations in the database or among tuples in a relation are embodied in the data itself, thus eliminating the need for external pointers or set relationships.

Data query and manipulation in the relational data model fall into the two general categories of relational algebra or relational calculus. The relational algebra expresses a query by applying a combination of special operations to some set of relations in the database. Typical algebraic operations include selection, projection, restriction, cross-product, division and the various joins (see [Tsichritzis,1977]). The relational calculus, on the other hand, is based on the predicate calculus where a query identifies a set of tuples that satisfy some specified condition or predicate. Although these two approaches differ significantly in concept and implementation, they each respond to a query by returning a relation (perhaps empty) that, hopefully, contains the desired information. The fact that a relation synthesized using one or more join operations from a relational algebra may contain spurious information is the central issue of this report and will be elaborated upon in the next chapter.

While the virtues of the relational data model such as

data independence, symmetry, flexibility and natural tabular form are well documented (see [Cardenas,1979]), it was noted by E. F. Codd as early as 1972 in [Codd,1972] that the relational model is not without its problems. In an attempt to free relational schemes from insertion, update and deletion anomalies, unnecessary redundancy and the need for restructuring as the database grows, increased levels of normalization were introduced. An excellent treatment of relational database normalization theory can be found in [Date,1981] or [Ullman,1982].

Chapter 2

THEORY

Certain problems with the relational data model such as data redundancy and update anomalies can be overcome by the decomposition of one or more relational schemes. The process of decomposing a relational scheme is simply one of replacing it with two or more sub-schemes where the set union of the attributes present in the sub-schemes is just the set of attributes in the original scheme. This solution, however, involves a trade-off as it is frequently necessary to recombine or "join" two or more schemes to answer a query and, in general, as the decomposition becomes more extensive the reliance on the join operation increases as well. The problem that arises is that for a given relational decomposition the natural join of two relations that represent the current values of two schemes may not always yield a relation containing only the information present before the decomposition was made. If some spurious information is generated in this process, the join is termed "lossy" and if not, the term "lossless" is used.

Since the decision to decompose a scheme into sub-schemes would normally be made during the design phase of the database implementation, it is highly desirable to

know whether or not a decomposition will lead to a "lossy" or "lossless" join before the database is actually implemented. The database administrator can gain this knowledge during the design phase by submitting the current decomposition of schemes to an implementation of Algorithm 7.2 on page 227 of [Ullman,1982].

The intent of this report is to describe an implementation of Algorithm 7.2 and not to be an expository work on the theory of joins therefore, most of the detail contained in the above reference will not be reproduced here. However, for the sake of completeness the algorithm will be included, along with a simple example of its use. Before the algorithm is stated it is necessary to define the term "functional dependency". An attribute B is said to be functionally dependent on an attribute A if, irrespective of time, the value of A determines the value of B, denoted by $A \rightarrow B$. The functional dependencies that hold for a particular relational scheme can only be determined by carefully considering the meanings of the attributes contained in the scheme.

The statement of Algorithm 7.2 that tests for a lossless join follows:

INPUT: A relation scheme $R=A_1 \ldots A_n$, a set of functional dependencies F, and a decomposition $d=(R_1,\ldots,R_k)$.

OUTPUT: A decision whether d is a decomposition with a lossless join.

METHOD: We construct a table with n columns and k rows; column j corresponds to attribute A_j, and row i corresponds to relation scheme R_i. In row i and column j put the symbol a_j if A_j is in R_i. If not, put the symbol b_{ij} there.

Repeatedly "consider" each of the dependencies X→Y in F, until no more changes can be made to the table. Each time we "consider" X→Y, we look for rows that agree in all the columns for the attributes of X. If we find two such rows, equate the symbols of those rows for the attributes of Y. When we equate two symbols, if one of them is a_j, make the other be a_j. If they are b_{ij} and b_{hj}, make them both b_{ij} or b_{hj}, arbitrarily.

If after modifying the rows of the table above, we discover that some row has become $a_1 \ldots a_n$, then the join is lossless. If not, the join is lossy (not lossless).

EXAMPLE: Consider the decomposition R1(AB), R2(AD), R3(AE), R4(BE) and R5(CDE) along with the functional dependencies A→C, B→CD, C→D, DE→C and CE→A. Table 2-1(a) shows the original table while Table 2-1(b) reveals the effect of the functional dependencies A→C and B→CD. Table 2-1(c) shows the effect of C→D and, finally, Table 2-1(d) reveals the effect of DE→C and CE→A. Note that the fourth row of Table 2-1(d) has become $a_1 .. a_5$. Therefore, this decomposition has the lossless join property with respect to the given set of functional dependencies.

A	B	C	D	E
a_1	a_2	b_{13}	b_{14}	b_{15}
a_1	b_{22}	b_{23}	a_4	b_{25}
a_1	b_{32}	b_{33}	b_{34}	a_5
b_{41}	a_2	b_{43}	b_{44}	a_5
b_{51}	b_{52}	a_3	a_4	a_5

(a)

A	B	C	D	E
a_1	a_2	b_{13}	b_{14}	b_{15}
a_1	b_{22}	b_{13}	a_4	b_{25}
a_1	b_{32}	b_{13}	b_{34}	a_5
b_{41}	a_2	b_{13}	b_{14}	a_5
b_{51}	b_{52}	a_3	a_4	a_5

(b)

A	B	C	D	E
a_1	a_2	b_{13}	a_4	b_{15}
a_1	b_{22}	b_{13}	a_4	b_{25}
a_1	b_{32}	b_{13}	a_4	a_5
b_{41}	a_2	b_{13}	a_4	a_5
b_{51}	b_{52}	a_3	a_4	a_5

(c)

A	B	C	D	E
a_1	a_2	b_{13}	a_4	b_{15}
a_1	b_{22}	b_{13}	a_4	b_{25}
a_1	b_{32}	a_3	a_4	a_5
a_1	a_2	a_3	a_4	a_5
a_1	b_{52}	a_3	a_4	a_5

(d)

Table 2-1

Chapter 3

DOCUMENTATION

GENERAL: This documentation refers to an implementation of Algorithm 7.2 on page 227 of Ullman, J. D. "Principles of Database Systems", Computer Science Press, Rockville, MD (2nd edition, 1982).

Given a relational decomposition and the accompanying set of functional dependencies this algorithm allows one to determine whether or not the particular decomposition has the "lossless join" property. Since the number of attributes, relations and functional dependencies will vary from decomposition to decomposition this implementation uses dynamic allocation of memory for those data structures that hold information concerning the decomposition or are required for the execution of the algorithm.

INTERNAL DOCUMENTATION: Algorithm 7.2 was implemented in Pascal for the PAS32 Compiler on the Interdata 8/32 computer under UNIX (version 7) at the Department of Computer Science, Kansas State University. The code for this implementation consists of a "prefix" followed by the Pascal program. The prefix contains CONST, TYPE, PROCEDURE and FUNCTION definitions that provide an interface between the compiler and the kernal of the operating system. The

number and order of the definitions in the prefix are an important feature of the interface and care must be exercised during program maintenance to insure that the prefix is not changed. The Pascal program consists of the six key procedures (readschema, readfds, buildtable, buildsearchlist, ullman, verdict), numerous auxiliary functions and procedures along with the usual complement of global definitions and declarations.

The internal documentation will begin with a discussion of the special input/output procedures required for the PAS32 compiler followed by a sort of "top down" discussion of the remaining subprograms in the general order in which they occur in the execution of the program. The data structures, global CONST and TYPE definitions and VAR declarations will also be discussed as they are encountered in the subprogram discussions.

Since PAS32 does not have the standard procedures read, write, readln and writeln implemented in the usual fashion it is necessary to include special procedures for other than character input/output. Thus, the procedures acceptint, displayint, displaystr, readint, writeint and writestr are included for integer input and integer or string output. The procedure acceptint accepts character input from a terminal one character (digit) at a time and generates usable integer fields in the range [0..32767]. The procedure readint is identical to acceptint except that it

reads character data from an external file instead of a terminal. The procedure displayint displays non-negative integer output to a terminal one character (digit) at a time and one integer field of up to eleven digits per line of output. The procedure writeint is identical to displayint except that it writes to an external file instead of a terminal. The procedure displaystr displays character strings as output to a terminal, one character at a time, until an end of line mark (:10:) is found. The procedure writestr is identical to displaystr except that it writes to an external file instead of a terminal. While this implementation only uses two of these procedures (writeint and writestr) the other four are included in the source code so that they will be readily available if needed during future program maintenance.

While this implementation is applicable to any relational decomposition, with its associated functional dependencies, it was designed so as to be compatible with the existing PAS32 implementation of Bernstein's Algorithm [Bernstein,1976]. Since this implementation of Bernstein's Algorithm accepts a set of functional dependencies as input and generates a third normal form schema as output it was imperative that the implementation of Algorithm 7.2 be able to read from two distinct input files, one containing the functional dependencies and the other containing the schema. Also, since PAS32 does not allow the simple designation of

an input file by naming it as the first argument in a read list the problem of input file designation is handled with the two procedures readfromfilel and readfromfile2. These procedures are identical except for the numeric file designation and could have been combined by passing the file number to a single procedure as a parameter. However, since these procedures are quite short it was decided to sacrifice programming style for efficiency and use a separate procedure to read from each file.

The procedure readfromfilel manipulates the global variables bcfilel, ccfilel and buffilel. These variables represent the block count, character count and input buffer for reading character data from filel. This procedure simply moves a 512 byte block from filel into buffilel and increments bcfilel whenever ccfilel reaches 513. If ccfilel is less than 513 a call to readfromfilel simply returns the character with index ccfilel in buffilel and increments ccfilel. The process is started by setting bcfilel to 1 and ccfilel to 513 initially in the main program. The procedure readfromfile2 behaves in similar fashion and needs no further comment. It is now possible to discuss the six key procedures contained in this implementation.

The procedure readschema is encountered first and as the name suggests it reads the relational schema from an external file and builds a data structure to hold the information obtained. This procedure is passed the global

variables headattrlist, relcount, and attrcount. Headattrlist is a pointer to the data structure built by readschema. Relcount and attrcount contain the count of relations and attributes in the schema, respectively. Readschema reads from the schema file a character at a time until an alpha character is read. It then calls procedure readattrfilel which continues to read characters from the schema file until the attribute is read and returns the attribute to readschema. Procedure listattr is now called and is passed the attribute just read along with the attribute count, relation count and a pointer to the data structure being created. If the attribute is new then it is assigned an integer code (the current value of attrct) and an attrcell is linked to the data structure along with a relcell showing the number of the relation containing this attribute. If a search reveals that the attribute is not new then only a relcell showing the relation number is linked to the data structure.

The procedure testwriteattr is a diagnostic output procedure used to assist in debugging and testing the procedure readschema. This procedure is passed the global variables headattrlist, relcount and attrcount. It writes the values of relcount and attrcount after which it navigates the data structure pointed to by headattrlist and writes each attribute stored along with its code and the numbers of the relations in which it can be found.

The procedure readfds reads the functional dependencies from an external file and builds a data structure to hold the information obtained. This procedure is passed the global variables headfdlist and headattrlist, where headfdlist is a pointer to the data structure built by readfds. Readfds reads from the functional dependency file a character at a time until an alpha character is read. It then calls procedure readattrfile2 which continues to read characters from the functional dependency file until the attribute is read and returns the attribute to readfds. If the attribute is not the endfdlist identifier (END.) and if not reading from a new functional dependency then the attribute code is obtained from the attribute list. This code and whether left or right side of dependency is placed in an fdattrcell and linked by procedure listattrcode to the data structure. If reading from a new functional dependency a new fdheadercell is linked before the fdattrcell.

The procedure testwritefds is a diagnostic output procedure used to assist in debugging and testing the procedure readfds. This procedure is passed the global variable headfdlist and it navigates the data structure pointed to by headfdlist and writes the attribute codes a functional dependency at a time. It also reveals whether a particular attribute is on the left hand or right hand side of the functional dependency in which it is located.

The procedure buildtable navigates the data structure

pointed to by the headattrlist and builds the table to be manipulated during the execution of Algorithm 7.2. This procedure is passed the global variables headtable, attrcount, relcount and headattrlist, where headtable, a result parameter, points to the cell in the first row and column of the table. The table has relcount rows and attrcount columns. Each cell of the table contains the row and column designation of its location, row and column pointer fields and an integer tag field. The tag field is set to zero if the attribute code (column designator) represents an attribute contained in the particular relation (row designator). Otherwise, the tag field is set to the row designator in preparation for the invocation of the algorithm.

The procedure testwritetable is a diagnostic output procedure used to assist in debugging and testing the procedure buildtable. This procedure is passed the global variable headtable and it navigates the table a row at a time and writes the values stored in the information fields of each cell. A description of each field written is also furnished.

The procedure buildsearchlist builds and initializes the data structure to be used to hold, and perhaps modify, the tag fields from the table during the execution of the algorithm. This procedure is passed the global variables headsearchlist and relcount, where the former is simply a

pointer to the data structure to be built by this procedure. The tag fields are arbitrarily initialized to the consecutive integer values from 1 to relcount which means that there is a cell in this structure for each row of the table. The two boolean fields in each cell are set to true. This initialization was done only to facilitate the testing of this procedure.

The procedure testwritesearchlist is a diagnostic output procedure used to assist in debugging and testing the procedure searchlist. This procedure is passed the global variable headsearchlist and it navigates the data structure pointed to by headsearchlist and writes the values stored in the tag, first and linked fields. A description of each field written is also furnished.

The procedure ullman is the heart of this implementation in that it considers each functional dependency in order and updates the table as required by Algorithm 7.2. This procedure is passed the global variables headtable, headfdlist, headsearchlist and tablechanged. As the name implies, tablechanged is a boolean variable whose value will be true if during any invocation of procedure ullman a change is made to one or more tag fields of the table. Ullman navigates the entire data structure pointed to by headfdlist, functional dependency by functional dependency, a cell at a time. If a new functional dependency is being considered, the data

structure searchlist is initialized with the boolean fields first and linked set to false and the pointer field nexttag set to nil. When a new cell of the data structure fdlist is visited the attribute code and whether left hand or right hand side of the dependency are determined. The tag fields of that column of the table corresponding to the current attribute code are now loaded into the tag fields of the structure searchlist by the local procedure loadtags. If the attribute code represents the first attribute on the left hand side of a functional dependency, then the nexttag pointer fields of the structure searchlist (for cells with equal tag fields) are linked by the local procedure linktags. If not the first attribute but still on the left hand side of a functional dependency, the nexttag pointer fields are relinked (current links may be broken) by the local procedure relinktags, if all tag fields within a previous linkage of nexttag pointer fields are not equal. It is important to note that if an attribute is on the left hand side of a functional dependency, only the nexttag pointer fields in the structure searchlist are manipulated and not the tag fields. If, on the other hand, the attribute code represents an attribute on the right hand side of a functional dependency, then the tag fields are again loaded from the table by the local procedure loadtags. At this point all tag fields within a given linked list of nexttag pointer fields are equated to the value of the

smallest tag in the list. In addition, all tags external to this linked list that had matching values with tags within the list are also set to this minimum value. After all tag field modifications have been made the modified tag fields are then returned to the column of the table corresponding to the current attribute code by the local procedure updatetable. If any change is made to a tag field of the structure searchlist, and thus ultimately to the table, a value of true is returned through the parameter list to the global variable tablechanged. If the value returned is false then the algorithm terminates.

The procedure verdict determines whether the given schema has the "lossless join" property. This procedure is passed the global variable tablehead and it navigates the table a row at a time. If any row has all tag fields equal to zero then the schema has the "lossless join" property and this fact is reported by writing an appropriate message to output.

The procedure writefinaltable is a diagnostic output procedure used to show the final condition of the table after the algorithm has terminated. This program is passed the global variables headtable and headattrlist. The tag fields of the table are written a column at a time until completed. However, the attribute corresponding to each column of the table is obtained from the structure attrlist and is written before the column is processed. Sample data

structures using the example at the end of Chapter 2 are contained in Appendix A.

EXTERNAL DOCUMENTATION: This segment of the documentation is devoted to the user perspective of the implementation of Algorithm 7.2. As such, it deals with the important issues of input file format, restrictions on identifiers, error messages and execution of the algorithm with its several options. It should be noted that this documentation presupposes user familiarity with the Interdata 8/32 computer under UNIX (version 7) in the Department of Computer Science at Kansas State University.

The source code and object modules for this implementation are contained in directory /usr/lib/projects/dbase and are protected against unauthorized modification. The identifier "ullman" is used to access this implementation in its various forms and on keying in "ullman" followed by a carriage return the system responds with:

```
usage: ullman [ -[afvt] [ -b fdfile] ] schemafile fdfile
```

This response reflects the five possible options available for invoking the algorithm.

The simplest option is invoked by keying in "ullman -v schemafile fdfile" followed by a carriage return, where schemafile and fdfile are input files containing the schema

for a relational database and its associated functional dependencies, respectively. The precise structure of these files will be specified later. This option executes only the output procedure verdict from the source program, after the algorithm terminates, and directs the output to the user terminal unless it is specifically redirected to an output file or device. This option will generate a row of output for each relation in the schemafile where a row of output consists of the string ********** or the string *******lossless*******. If the latter string appears one or more times in the output then the given schema has the lossless join property.

The next option is invoked by keying in "ullman -t schemafile fdfile" followed by a carriage return. This option is the same as "ullman -v" above except that the final table generated by the algorithm is also written to output by the output procedure writefinaltable as described in the internal documentation.

The next option is invoked by keying in "ullman -f schemafile fdfile" followed by a carriage return. This option is the same as "ullman -t" above except that the information fields of the data structure containing the functional dependencies are also written to output by the output procedure testwritefds as described in the internal documentation.

The last option using two input files is invoked by

keying in "ullman -a schemafile fdfile" followed by a carriage return. This option is the same as "ullman -f" above except that the information fields of the data structure containing the schema are also written to output by the output procedure testwriteattr as described in the internal documentation. In addition, the output procedure testwritetable is invoked after procedure buildtable and again after each pass of the algorithm.

The final option requires only one input file and is invoked by keying in "ullman -b fdfile" followed by a carriage return. Each of the following tasks described is accomplished automatically and no action is required on the part of the user. This option differs significantly from the previous ones in that the fdfile is first submitted as input to "bern5", the Kansas State University implementation of Bernstein's Algorithm. The third normal form schema is extracted from the output from "bern5" and is placed in a newly created file called "schemafile" that can now be found in the current directory. The end of medium character (control-Y) has been affixed to the end of schemafile as required by the source program for proper input control. At this point both the schemafile and fdfile are submitted to "ullman -v" with the same effect as already described above, except that the schemafile is written to output before the verdict is announced. If one wants to use "bern5" along with an option other than "ullman -v" it is a simple matter

to invoke "ullman -b" and then to invoke "ullman" with any of the other three options using as input the original fdfile and the newly generated schemafile now located in the current directory.

The formats of the schemafile and fdfile were dictated by the existing implementation "bern5" since one option of "ullman" uses both the input to and partial output from "bern5" as its two input sources. An attribute name in either file may be at most 28 characters in length and must begin with a letter (upper or lower case) followed by zero or more letters, digits, %, _, or # (except END). If option "ullman -b" is used then the attributes may not contain lower case letters or special characters as "bern5" does not recognize them.

The structure of the fdfile is a rather natural representation of a set of functional dependencies and is probably best described by the following example:

A → C	A > C ;
B → CE	B > C , E ;
C → D	C > D ;
DE → C	D , E > C ;
CE → A	C , E > A ;
	END. ^Y

Example 3-1

In this example the left hand column represents a set of six functional dependencies with single character attributes "A" through "E" as they might appear in the database literature. The right hand column reveals the corresponding fdfile format for this set of dependencies. Since the semicolon is used as a separator, imbedded blanks are ignored and more than one dependency may be placed on each line if desired. Regardless of which option of "ullman" is used it is essential that the string "END" in the last line of the file be upper case.

Although the structure of the schemafile is quite unlike the usual representation of a relational schema it is probably still best described by the following example:

```
R1(A,C,E,D)          (D E ) (C E ) > A
R2(C,D)              (C ) > D
R3(A,C)              (A ) > C
R4(B,C,E)            (B ) > C E
                     ^Y
```

Example 3-2

In this example the right hand column reveals the structure of the schemafile generated by submitting the fdfile of Example 3-1 to "ullman -b". As was noted earlier, this file is only a part of the output from "bern5" and represents a

third normal form schema for the relational database with the set of functional dependencies as shown in Example 3-1. Each row of this particular file represents a relation in the schema that contains each of the attributes shown in that row. The attributes in parentheses represent candidate keys for that relation whereas, non-prime attributes appear to the right of the greater-than sign. The left hand column of this example shows a set of relations as they might appear in the database literature. Since "ullman" is applicable to other than third normal form schema it is not necessary to identify the candidate keys of a relation. Hence, the schemafile for the relational schema of Example 3-2 could be simply constructed as follows:

```
(A C D E) >
(C D) >
(A C) >
(B C E) >
^Y
```

Example 3-3

Obviously, this would only be done if an option other than "ullman -b" were being used, as "ullman -b" automatically creates the schemafile for the user. If the schemafile is being created by the user, then it is crucial that at least

one pair of parentheses appear to the left of the greater-than sign in each relation, as the first left-parenthesis encountered after a greater-than sign has been read signals the beginning of a new relation. Also, recall that schemafile must terminate with a (control-Y).

The source program for this implementation of Algorithm 7.2 emits only five error messages. There are two in the procedure readschema and three in the procedure readfds. If the first character read in the procedures readschema or readfds is an end of medium character (control-Y), then the program responds with "**error-schema file empty**" or "**error-fd file empty**", respectively. If a character is read from the schemafile or fdfile that is not in the respective CASE constant lists of procedures readschema or readfds, then a response of "**error-invalid character in schema file**" or "**error-invalid character in fd file**" is received. It should be noted that the failure to terminate either input file with (control-Y) will result in an infinite loop with one of the two previous error messages repeatedly written to output. The function attrcode is local to procedure readfds and will respond with the error message "**error-attribute not found**" if it cannot find the attribute, passed to it as a parameter, in the current list of attributes. This message will be received only if the fdfile contains an attribute not encountered in the schemafile. A simple misspelling of an attribute name to

include an upper-lower case mismatch is the likely cause.

It is important to note that each option of "ullman" is independent of the others and, therefore, multiple options can not be used.

Chapter 4

CONCLUSIONS

This implementation should be a valuable addition to the growing family of departmental database utility programs as its five options allow for a wide variety of uses. For example, if one is designing a relational database and has only a set of functional dependencies, then option "ullman -b" will generate a set of third normal form schemes as well as determine whether or not the decomposition has the lossless join property. If a decomposition of schemes has already been made, then option "ullman -v" will simply check for lossless joins. If, on the other hand, one is interested in following the execution of the algorithm, then option "ullman -a" will write the modified table to output after each pass through the functional dependencies.

Future work in the relational database area might include an implementation of an algorithm for testing the preservation of functional dependencies with respect to a given relational decomposition. This could be followed by an implementation of a lossless join algorithm for multivalued dependencies.

Appendix A

DATA STRUCTURES

Cell Descriptions

attrcell (attrlist)

attrname	attrcode	nextattr	inrel

relcell (attrlist)

relnumber	nextrelnum

fdheadercell (fdlist)

nextfd	nextfdattr

fdattrcell (fdlist)

attrcode	lhs	nextfdattr

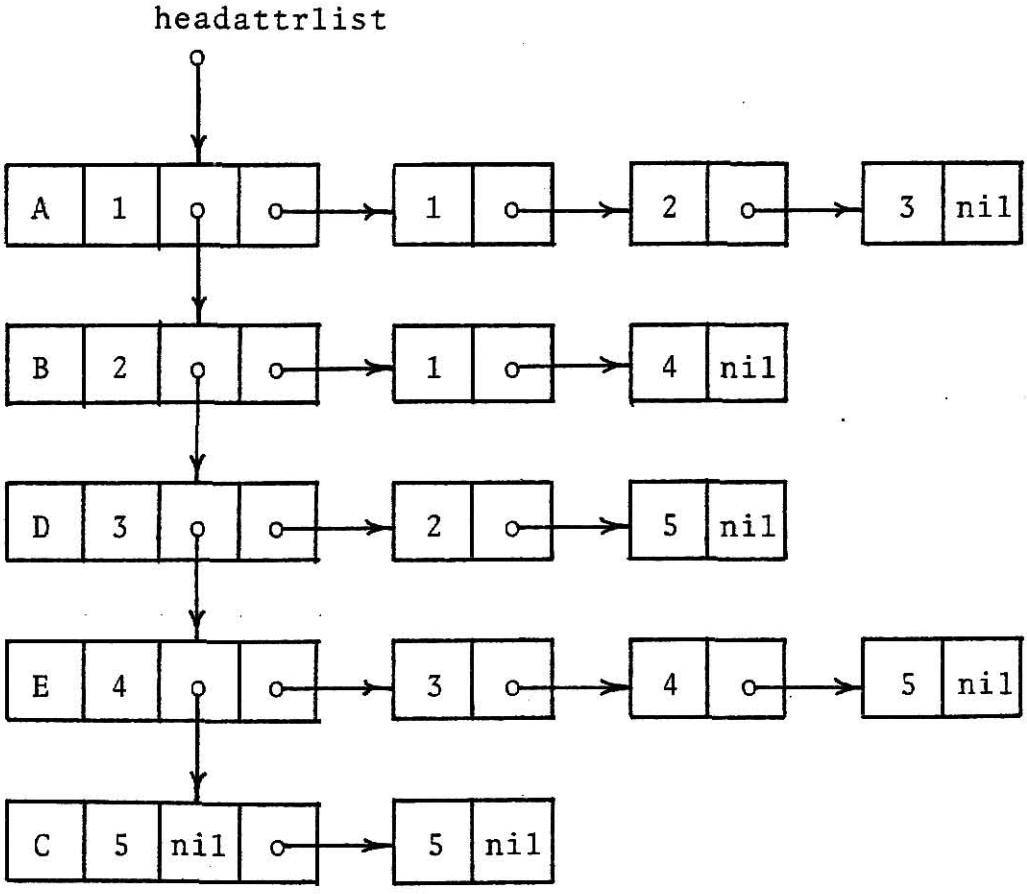
tablecell (table)

attrcode	
row	rowptr
colptr	tag

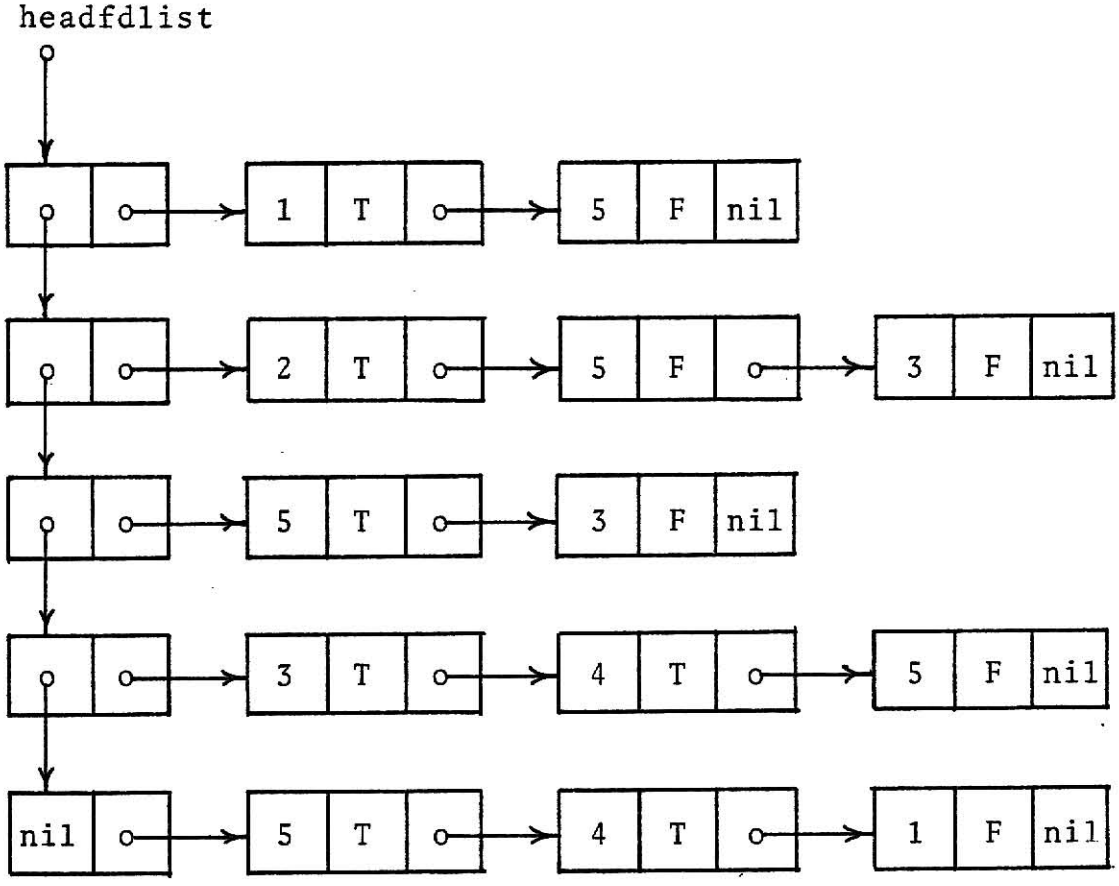
searchcell (searchlist)

nextcell	tag	linked	first	nexttag

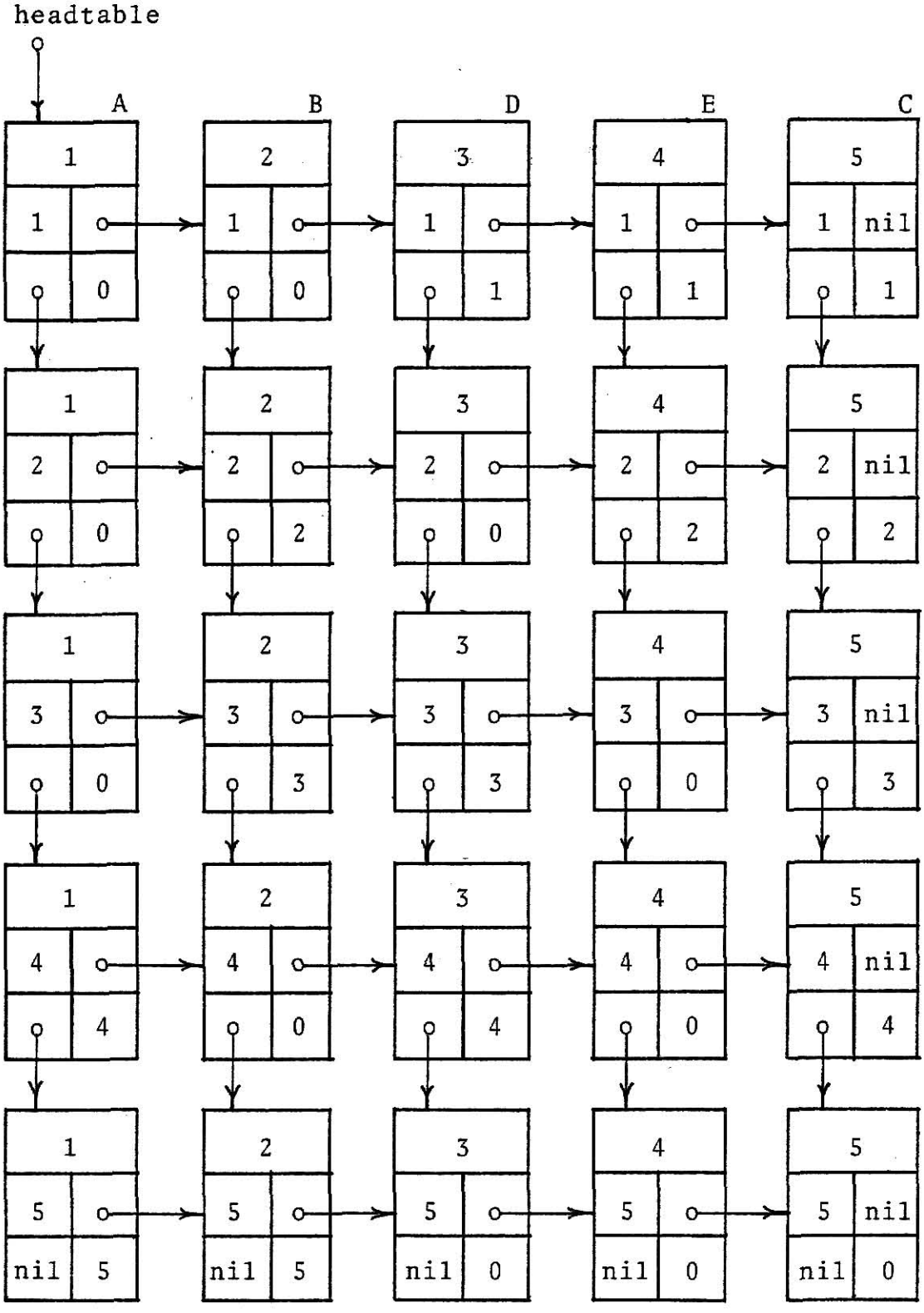
Attribute List



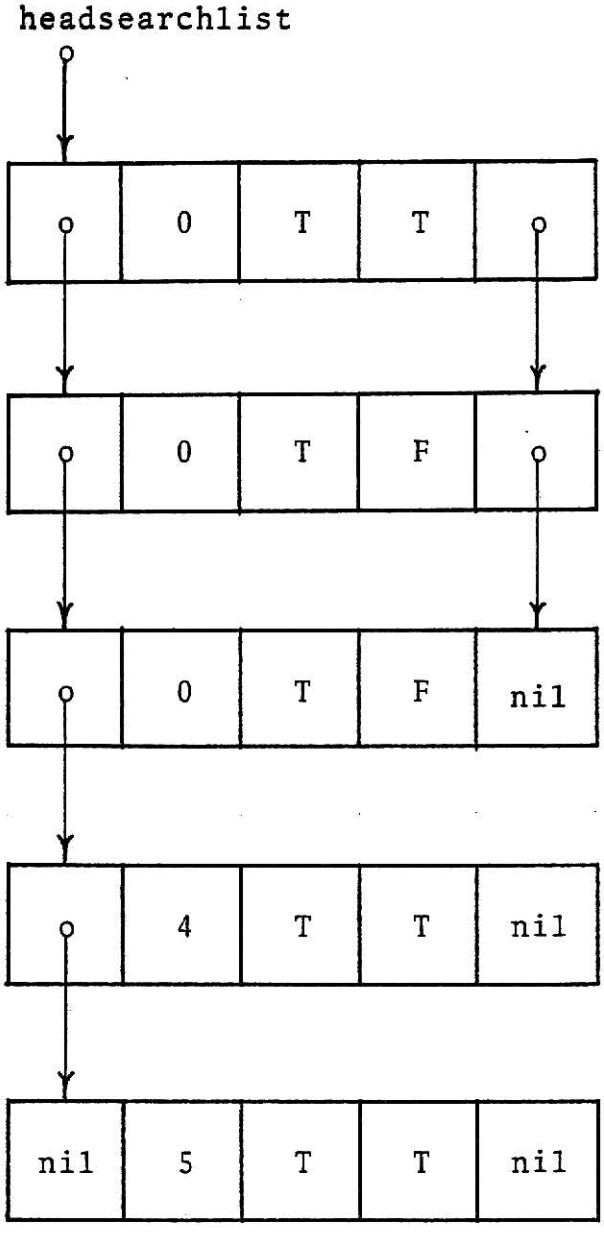
Functional Dependency List



Initial Table



Search List



Appendix B

SOURCE CODE

```
"per brinch hansen                    *   as modified for the interdata
                                      *     8/32 under unix v7 at the
 information science                  *   department of computer science
 california institute of technology   *      kansas state university
                                      *
 utility programs for                 *
 the solo system                      *
                                      *
 18 may 1975                          *            1 jan 1983"

"###########
#  prefix  #
###########"

  const nl = '(:10:)'
  ; ff = '(:12:)'
  ; cr = '(:13:)'
  ; em = '(:25:)'

; const pagelength = 512

; type page = array (. 1 .. pagelength .) of char

; const linelength = 132

; type line = array (. 1 .. linelength .) of char

; const idlength = 12

; type identifier = array (. 1 .. idlength .) of char

; type file = 1 .. 2

; type filekind = ( empty , scratch , ascii , seqcode , concode )

; type fileattr
```

```
    = record
        kind : filekind
      ; addr : integer
      ; protected : boolean
      ; notused : array (. 1 .. 5 .) of integer
      end

; type iodevice
    = ( typedevice , diskdevice , tapedevice , printdevice
      , carddevice )

; type iooperation = ( input , output , move , control )

; type ioarg = ( writeeof , rewind , upspace , backspace )

; type ioresult
    = ( complete , intervention , transmission , failure , endfile
      , endmedium , startmedium )

; type ioparam
    = record
        operation : iooperation
      ; status : ioresult
      ; arg : ioarg
      end

; type taskkind = ( inputtask , jobtask , outputtask )

; type argtag = ( niltype , booltype , inttype , idtype , ptrtype )

; type pointer = @ boolean

; type argtype
    = record
        case tag : argtag
        of niltype , booltype : ( bool : boolean )
        ; inttype : ( int : integer )
        ; idtype : ( id : identifier )
        ; ptrtype : ( ptr : pointer )
      end

; const maxarg = 10

; type arglist = array (. 1 .. maxarg .) of argtype

; type argseq = ( inp , out )

; type progresult
    = ( terminated , overflow , pointererror , rangeerror
      , varianterror , heaplimit , stacklimit , codelimit , timelimit
      , callerror )
```

```
; procedure read ( var c : char )

; procedure write ( c : char )

; procedure open ( f : file ; id : identifier ; var found : boolean )

; procedure close ( f : file )

; procedure get ( f : file ; p : integer ; var block : univ page )

; procedure put ( f : file ; p : integer ; var block : univ page )

; function length ( f : file ) : integer

; procedure mark ( var top : integer )

; procedure release ( top : integer )

; procedure identify ( header : line )

; procedure accept ( var c : char )

; procedure display ( c : char )

; procedure readpage ( var block : univ page ; var eof : boolean )

; procedure writepage ( block : univ page ; eof : boolean )

; procedure readline ( var text : univ line )

; procedure writeline ( text : univ line )

; procedure readarg ( s : argseq ; var arg : argtype )

; procedure writearg ( s : argseq ; arg : argtype )

; procedure lookup
    ( id : identifier ; var attr : fileattr ; var found : boolean )

; procedure iotransfer
    ( device : iodevice
    ; var param : ioparam
    ; var block : univ page
    )

; procedure iomove ( device : iodevice ; var param : ioparam )

; function task : taskkind

; procedure run
    ( id : identifier
    ; var param : arglist
```

```
    ; var line : integer
    ; var result : progresult
    )

"#################
#  end of prefix  #
#################"

(*****************************************************************************)
(*****************************************************************************)
(**                                                                         **)
(** this program implements algorithm 7.2 on page 227 of Ullman, J.        **)
(** PRINCIPLES OF DATABASE SYSTEMS, Computer Science Press,                 **)
(** Rockville,MD (2nd edition, 1982).                                       **)
(**                                                                         **)
(** written by: karl klose                                                  **)
(**                                                                         **)
(** date: may 1983                                                          **)
(**                                                                         **)
(*****************************************************************************)
(*****************************************************************************)

; program p ( paramline : line )

  ; const maxattrlength = 28

  ; type attrary = packed array [ 1 .. maxattrlength ] of char
    ; attrptr = @ attrcell
    ; relptr = @ relcell
    ; fdhptr = @ fdheadercell
    ; fdptr = @ fdattrcell
    ; tblptr = @ tablecell
    ; srchptr = @ searchcell
    ; attrcell
      = record
          attrname : attrary
        ; attrcode : integer
        ; nextattr : attrptr
        ; inrel : relptr
        end
    ; relcell = record relnumber : integer ; nextrelnum : relptr end
    ; fdheadercell = record nextfd : fdhptr ; nextfdattr : fdptr end
    ; fdattrcell
      = record
          attrcode : integer
        ; lhs : boolean
        ; nextfdattr : fdptr
        end
    ; tablecell
      = record
          attrcode : integer
        ; row : integer
```

```
        ; tag : integer
        ; rowptr : tblptr
        ; colptr : tblptr
        end
    ; searchcell
      = record
          tag : integer
        ; linked : boolean
        ; first : boolean
        ; nextcell : srchptr
        ; nexttag : srchptr
        end

  ; var ccfile1 : integer
    ; ccfile2 : integer
    ; bcfile1 : integer
    ; bcfile2 : integer
    ; buffile1 : page
    ; buffile2 : page
    ; headattrlist : attrptr
    ; relcount : integer
    ; attrcount : integer
    ; headfdlist : fdhptr
    ; headtable : tblptr
    ; headsearchlist : srchptr
    ; tablechanged : boolean

(*************************************************************************)
(* this procedure accepts character input from a terminal one char-      *)
(* acter at a time and generates usable integer fields.                  *)
(*************************************************************************)

  ; procedure acceptint ( var int : integer )

    ; const maxint = 32767

    ; type digit = '0' .. '9'

    ; var overflow : boolean
      ; d : integer
      ; digits : set of digit
      ; c : char

    ; begin
        int := 0
      ; overflow := false
      ; digits := [ ]
      ; for c := '0' to '9' do digits := digits + [ c ]
      ; accept ( c )
      ; while ( ( c in digits ) & not overflow )
        do begin
          d := ord ( c ) - ord ( '0' )
```

```
        ; if int > ( maxint - d ) div 10
          then overflow := true
          else int := 10 * int + d
        ; accept ( c )
        ;
        end
      ; if overflow
        then begin
          int := maxint
        ; while ( c in digits ) do accept ( c )
        ;
        end
      end (*procedure acceptint*)

(*************************************************************************)
(* this procedure displays integer output one character at a time and    *)
(* one field per line of output                                          *)
(*************************************************************************)

  ; procedure displayint ( i : integer )

    ; var digit , j , int : integer
      ; intstring : array [ 1 .. 11 ] of char

    ; begin
        int := abs ( i )
      ; for j := 1 to 11 do intstring [ j ] := ' '
      ; j := 11
      ; repeat digit := int mod 10
        ; intstring [ j ] := chr ( digit + 48 )
        ; int := int div 10
        ; j := j - 1
        until int = 0
      ; if i < 0
        then intstring [ j ] := '-'
      ; for j := 1 to 11 do display ( intstring [ j ] )
      ; display ( nl )
      end (*procedure displayint*)

(*************************************************************************)
(* this procedure displays character strings as output, one character    *)
(* at a time until an end of line mark (:10:) is found.                  *)
(*************************************************************************)

  ; procedure displaystr ( text : line )

    ; var i : integer

    ; begin
        i := 1
      ; while ( text [ i ] <> '(:10:)' )
        do begin
```

```
          display ( text [ i ] )
        ; i := i + 1
        end
      ; display ( nl )
      end (*procedure displaystr*)

(*************************************************************************)
(* this procedure reads character input one character at a time and      *)
(* generates useable integer fields.                                     *)
(*************************************************************************)

  ; procedure readint ( var int : integer )

    ; const maxint = 32767

    ; type digit = '0' .. '9'

    ; var overflow : boolean
      ; d : integer
      ; digits : set of digit
      ; c : char

    ; begin
        int := 0
      ; overflow := false
      ; digits := [ ]
      ; for c := '0' to '9' do digits := digits + [ c ]
      ; read ( c )
      ; while ( ( c in digits ) & not overflow )
        do begin
          d := ord ( c ) - ord ( '0' )
        ; if int > ( maxint - d ) div 10
          then overflow := true
          else int := 10 * int + d
        ; read ( c )
        ;
        end
      ; if overflow
        then begin
          int := maxint
        ; while ( c in digits ) do read ( c )
        ;
        end
      end (*procedure readint*)

(*************************************************************************)
(* this procedure writes integer output one character at a time and      *)
(* one field per line of output.                                         *)
(*************************************************************************)

  ; procedure writeint ( i : integer )
```

```
    ; var digit , j , int : integer
      ; intstring : array [ 1 .. 11 ] of char

    ; begin
        int := abs ( i )
      ; for j := 1 to 11 do intstring [ j ] := ' '
      ; j := 11
      ; repeat digit := int mod 10
        ; intstring [ j ] := chr ( digit + 48 )
        ; int := int div 10
        ; j := j - 1
        until int = 0
      ; if i < 0
        then intstring [ j ] := '-'
      ; for j := 1 to 11 do write ( intstring [ j ] )
      ; write ( nl )
      end (*procedure writeint*)

(*************************************************************************)
(* this procedure writes character strings as output, one character at *)
(* a time until an end of line mark (:10:) is found.                     *)
(*************************************************************************)

  ; procedure writestr ( text : line )

    ; var i : integer

    ; begin
        i := 1
      ; while ( text [ i ] <> '(:10:)' )
        do begin
          write ( text [ i ] )
        ; i := i + 1
        end
      ; write ( nl )
      end (*procedure writestr*)

(*************************************************************************)
(* this procedure manipulates the global variables bcfile1, ccfile1     *)
(* and buffile1.  it simply moves a 512 byte block from file1 into      *)
(* the buffer buffile1 where it is read a character at a time until     *)
(* exhausted, at which time the next block is moved to buffile1.        *)
(* bcfile1 is the block count and is set to 1 initially, whereas        *)
(* ccfile1 is the character count and is set to 513 initially.          *)
(*************************************************************************)

  ; procedure readfromfile1 ( var c : char )

    ; begin
        if ccfile1 > 512
        then begin
          ccfile1 := 1
```

```
        ; get ( 1 , bcfile1 , buffile1 )
        ; bcfile1 := succ ( bcfile1 )
        ;
        end
      ; c := buffile1 [ ccfile1 ]
      ; ccfile1 := succ ( ccfile1 )
      ;
      end (*procedure readfromfile1*)

(*************************************************************************)
(* this procedure manipulates the global variables bcfile2, ccfile2     *)
(* and buffile2.  it simply moves a 512 byte block from file2 into      *)
(* the buffer buffile2 where it is read a character at a time until     *)
(* exhausted, at which time the next block is moved to buffile2.        *)
(* bcfile2 is the block count and is set to 1 initially, whereas        *)
(* ccfile2 is the character count and is set to 513 initially.          *)
(*************************************************************************)

  ; procedure readfromfile2 ( var c : char )

    ; begin
        if ccfile2 > 512
        then begin
          ccfile2 := 1
        ; get ( 2 , bcfile2 , buffile2 )
        ; bcfile2 := succ ( bcfile2 )
        ;
        end
      ; c := buffile2 [ ccfile2 ]
      ; ccfile2 := succ ( ccfile2 )
      ;
      end (*procedure readfromfile2*)

(*************************************************************************)
(* once an alpha character is found by the calling routine this pro-    *)
(* cedure continues reading characters from file1 and forming an at-    *)
(* tribute name until a non-alpha character is read.  the attribute     *)
(* name and the non-alpha character are then returned to the calling    *)
(* routine.                                                             *)
(*************************************************************************)

  ; procedure readattrfile1 ( var attr : attrary ; var ch : char )

    ; var index : 1 .. maxattrlength
      ; count : 1 .. maxattrlength
      ; goodcharset : set of char

    ; begin
        goodcharset
        := [ 'a' .. 'z' , 'A' .. 'Z' , '0' .. '9' , '#' , '_' , '%' ]
      ; for index := 1 to maxattrlength do attr [ index ] := ' '
      ; count := 1
```

```
      ; while ( ch in goodcharset )
        do begin
          attr [ count ] := ch
        ; count := count + 1
        ; readfromfile1 ( ch )
        end
      end (*procedure readattrfile1*)

(*************************************************************************)
(* once a alpha character is found by the calling routine this pro-      *)
(* cedure continues reading characters from file2 and forming an at-     *)
(* tribute name until a non-alpha character is read.  the attribute      *)
(* name and the non-alpha character are then returned to the calling     *)
(* routine.                                                              *)
(*************************************************************************)

  ; procedure readattrfile2 ( var attr : attrary ; var ch : char )

    ; var index : 1 .. maxattrlength
      ; count : 1 .. maxattrlength
      ; goodcharset : set of char

    ; begin
        goodcharset
        := [ 'a' .. 'z' , 'A' .. 'Z' , '0' .. '9' , '#' , '_' , '%' ]
      ; for index := 1 to maxattrlength do attr [ index ] := ' '
      ; count := 1
      ; while ( ch in goodcharset )
        do begin
          attr [ count ] := ch
        ; count := count + 1
        ; readfromfile2 ( ch )
        end
      end (*procedure readattrfile2*)

(*************************************************************************)
(* this procedure receives an attribute just read, a current count of    *)
(* relations, a current count of attributes and the head-pointer of      *)
(* the attribute list and either finds the attribute in the list and     *)
(* updates its relation number or enters the new attribute in the list   *)
(* along with its relation number.  the attribute count and head-        *)
(* pointer of the list are returned to the calling routine.              *)
(*************************************************************************)

  ; procedure listattr
      ( attr : attrary
      ; relct : integer
      ; var attrct : integer
      ; var headptr : attrptr
      )

    ; var aptr : attrptr
```

```
    ; rptr : relptr
    ; reltail : relptr
    ; location : attrptr
    ; found : boolean

(*************************************************************************)
(* this procedure is local to procedure listattr and simply gener-       *)
(* ates, links and defines a new attribute record when the local         *)
(* procedure searchattrlist does not find a given attribute.  it         *)
(* also increments the attribute count and returns this value to         *)
(* the calling routine.                                                  *)
(*************************************************************************)

  ; procedure listnewattr
      ( attr : attrary
      ; aptr : attrptr
      ; var attrct : integer
      ; relct : integer
      )

    ; var rptr : relptr
      ; index : integer
      ; atemp : attrptr

    ; begin
        attrct := attrct + 1
      ; atemp := aptr
      ; for index := 1 to maxattrlength
        do atemp @ . attrname [ index ] := attr [ index ]
      ; atemp @ . attrcode := attrct
      ; atemp @ . nextattr := nil
      ; new ( rptr )
      ; atemp @ . inrel := rptr
      ; rptr @ . relnumber := relct
      ; rptr @ . nextrelnum := nil
      end (*procedure listnewattr*)

(*************************************************************************)
(* this procedure is local to procedure listattr and simply searches     *)
(* the non-empty attribute list and returns the location of the at-      *)
(* tribute, if found, or the tail of the attribute list if the at-       *)
(* tribute is not found.                                                 *)
(*************************************************************************)

  ; procedure searchattrlist
      ( head : attrptr
      ; attr : attrary
      ; var location : attrptr
      ; var found : boolean
      )

    ; var temp : attrptr
```

```
    ; begin
        temp := head
      ; while ( ( temp @ . attrname <> attr )
                & ( temp @ . nextattr <> nil )
              )
        do temp := temp @ . nextattr
      ; if temp @ . attrname = attr
        then begin found := true ; location := temp end
        else begin found := false ; location := temp end
      end (*procedure searchattrlist*)

(*************************************************************************)
(* this function is local to procedure listattr and is invoked only  *)
(* when the attribute just read is found in the attribute list.  in  *)
(* this case it simply scans the relation incidence list for this    *)
(* attribute and returns the location of the tail of this list to    *)
(* the calling routine.                                              *)
(*************************************************************************)

  ; function rtail ( location : attrptr ) : relptr

    ; var temp : relptr

    ; begin
        temp := location @ . inrel
      ; while ( temp @ . nextrelnum <> nil )
        do temp := temp @ . nextrelnum
      ; rtail := temp
      end (*function rtail*)

  ; begin (*procedure listattr*)
      if headptr = nil
      then begin
        new ( aptr )
      ; headptr := aptr
      ; listnewattr ( attr , aptr , attrct , relct )
      end
      else begin
        searchattrlist ( headptr , attr , location , found )
      ; if not found
        then begin
          new ( aptr )
        ; location @ . nextattr := aptr
        ; listnewattr ( attr , aptr , attrct , relct )
        end
        else begin
          new ( rptr )
        ; reltail := rtail ( location )
        ; reltail @ . nextrelnum := rptr
        ; rptr @ . relnumber := relct
        ; rptr @ . nextrelnum := nil
```

```
        end
      end
    end (*procedure listattr*)

(*************************************************************************)
(* this procedure reads the relational schema from a file and creates    *)
(* a linked structure of attributes found and relations in which they    *)
(* are found.  the pointer to the head of this structure, the number     *)
(* of relations found, and the number of attributes found are returned   *)
(* to the calling routine.                                               *)
(*************************************************************************)

  ; procedure readschema
      ( var headptr : attrptr ; var relct , attrct : integer )

    ; var attr : attrary
      ; arrow : boolean
      ; ch : char

    ; begin
        arrow := true
      ; relct := 0
      ; attrct := 0
      ; headptr := nil
      ; readfromfile1 ( ch )
      ; if ch = em
        then writestr ( '**error-schema file empty**(:10:)' )
        else while ( ch <> em )
          do case ch
            of 'A' .. 'Z' , 'a' .. 'z'
              : begin
                readattrfile1 ( attr , ch )
              ; listattr ( attr , relct , attrct , headptr )
              end
            ; nl , ' ' , ')' : readfromfile1 ( ch )
            ; '('
              : begin
                if arrow
                then begin
                  arrow := false
                ; relct := relct + 1
                ; readfromfile1 ( ch )
                end
                else readfromfile1 ( ch )
              end
            ; '>' : begin arrow := true ; readfromfile1 ( ch ) end
            ; else
              : begin
                write ( ch )
              ; writestr
                  ( '**error-invalid character in schema file**(:10:)'
                  )
```

```
            ; readfromfile1 ( ch )
            end
          end
      end (*procedure readschema*)

(*************************************************************************)
(* this procedure writes the salient information stored in the data      *)
(* structure constructed by the procedure readschema.                    *)
(*************************************************************************)

  ; procedure testwriteattr ( head : attrptr ; rct , act : integer )

    ; var atemp : attrptr
      ; rtemp : relptr
      ; i : integer

    ; begin
        writestr ( '****enter procedure testwriteattr****(:10:)' )
      ; writestr ( 'the rel ct is(:10:)' )
      ; writeint ( rct )
      ; writestr ( 'the attr ct is(:10:)' )
      ; writeint ( act )
      ; atemp := head
      ; repeat for i := 1 to maxattrlength
          do write ( atemp @ . attrname [ i ] )
        ; write ( nl )
        ; writestr ( 'attr code is(:10:)' )
        ; writeint ( atemp @ . attrcode )
        ; writestr ( 'attr is in following rel(:10:)' )
        ; rtemp := atemp @ . inrel
        ; repeat writeint ( rtemp @ . relnumber )
          ; rtemp := rtemp @ . nextrelnum
          until ( rtemp = nil )
        ; atemp := atemp @ . nextattr
        until ( atemp = nil )
      ; writestr ( '****leave procedure testwriteattr****(:10:)' )
      end (*procedure testwriteattr*)

(*************************************************************************)
(* this procedure reads the functional dependencies from a file and      *)
(* creates a linked structure that contains each functional depend-      *)
(* ency.                                                                 *)
(*************************************************************************)

  ; procedure readfds
      ( var fdheadptr : fdhptr ; attrheadptr : attrptr )

    ; var attr : attrary
      ; ch : char
      ; lhs : boolean
      ; newfdlevel : boolean
      ; endfdlist : attrary
```

```
    ; index : integer
    ; acode : integer
    ; tempptr : fdhptr

(*************************************************************************)
(* this procedure is local to procedure readfds and simply creates       *)
(* and links a new header cell to the structure before a new level       *)
(* of functional dependencies is read and linked.                        *)
(*************************************************************************)

  ; procedure linknewhdrcell ( var fdheadptr , tempptr : fdhptr )

    ; var hdrptr : fdhptr

    ; begin
        if fdheadptr = nil
        then begin
          new ( hdrptr )
        ; fdheadptr := hdrptr
        ; tempptr := hdrptr
        ; tempptr @ . nextfd := nil
        ; tempptr @ . nextfdattr := nil
        end
        else begin
          new ( hdrptr )
        ; tempptr @ . nextfd := hdrptr
        ; tempptr := hdrptr
        ; tempptr @ . nextfd := nil
        ; tempptr @ . nextfdattr := nil
        end
      end (*procedure linknewhdrcell*)

(*************************************************************************)
(* this function is local to procedure readfds and when passed an        *)
(* attribute it returns the corresponding attribute code.                *)
(*************************************************************************)

  ; function attrcode ( attr : attrary ; ahptr : attrptr ) : integer

    ; var tempaptr : attrptr
      ; index : integer

    ; begin
        tempaptr := ahptr
      ; while ( ( tempaptr @ . attrname <> attr )
                & ( tempaptr <> nil )
              )
        do tempaptr := tempaptr @ . nextattr
      ; if tempaptr = nil
        then begin
          writestr ( '**error-attribute not found**(:10:)' )
        ; for index := 1 to maxattrlength
```

```
        do write ( attr [ index ] )
      ; attrcode := 0
      end
      else attrcode := tempaptr @ . attrcode
    end (*function attrcode*)

(*************************************************************************)
(* this procedure is local to procedure readfds and it creates,      *)
(* defines and links a new cell in the functional dependency linked  *)
(* structure to hold the attribute code and location of the current  *)
(* attribute.                                                        *)
(*************************************************************************)

  ; procedure listattrcode
      ( tempptr : fdhptr ; acode : integer ; lhs : boolean )

    ; var cellptr : fdptr
      ; next : fdptr
      ; temp : fdhptr

    ; begin
        if tempptr @ . nextfdattr = nil
        then begin
          new ( cellptr )
        ; temp := tempptr
        ; temp @ . nextfdattr := cellptr
        ; cellptr @ . attrcode := acode
        ; cellptr @ . lhs := lhs
        ; cellptr @ . nextfdattr := nil
        end
        else begin
          next := tempptr @ . nextfdattr
        ; while ( next @ . nextfdattr <> nil )
          do next := next @ . nextfdattr
        ; new ( cellptr )
        ; next @ . nextfdattr := cellptr
        ; cellptr @ . attrcode := acode
        ; cellptr @ . lhs := lhs
        ; cellptr @ . nextfdattr := nil
        end
      end (*procedure listattrcode*)

  ; begin (*procedure readfds*)
      lhs := true
    ; newfdlevel := true
    ; fdheadptr := nil
    ; endfdlist [ 1 ] := 'E'
    ; endfdlist [ 2 ] := 'N'
    ; endfdlist [ 3 ] := 'D'
    ; for index := 4 to maxattrlength do endfdlist [ index ] := ' '
    ; readfromfile2 ( ch )
    ; if ch = em
```

```
        then writestr ( '**error-fd file empty**(:10:)' )
        else while ( ch <> em )
          do case ch
            of 'A' .. 'Z' , 'a' .. 'z'
              : begin
                readattrfile2 ( attr , ch )
              ; if attr <> endfdlist
                then begin
                  if newfdlevel
                  then begin
                    linknewhdrcell ( fdheadptr , tempptr )
                  ; newfdlevel := false
                  end
                ; acode := attrcode ( attr , attrheadptr )
                ; listattrcode ( tempptr , acode , lhs )
                end
            ; '>' : begin lhs := false ; readfromfile2 ( ch ) end
            ; nl , ' ' , ',' , '.' : readfromfile2 ( ch )
            ; ';'
              : begin
                lhs := true
              ; newfdlevel := true
              ; readfromfile2 ( ch )
              end
            ; else
              : begin
                write ( ch )
              ; writestr
                  ( '**error-invalid character in fd file**(:10:)' )
              ; readfromfile2 ( ch )
              end
          end
    end (*procedure readfds*)

(*************************************************************************)
(* this procedure writes the salient information stored in the data      *)
(* structure constructed by the procedure readfds.                       *)
(*************************************************************************)

  ; procedure testwritefds ( head : fdhptr )

    ; var hdrtemp : fdhptr
      ; celltemp : fdptr

    ; begin
        writestr ( '****enter procedure testwritefds****(:10:)' )
      ; hdrtemp := head
      ; repeat celltemp := hdrtemp @ . nextfdattr
        ; repeat writeint ( celltemp @ . attrcode )
          ; if celltemp @ . lhs
            then writestr ( '*is on the LHS*(:10:)' )
```

```
            else writestr ( '*is on the RHS*(:10:)' )
          ; celltemp := celltemp @ . nextfdattr
          until ( celltemp = nil )
        ; hdrtemp := hdrtemp @ . nextfd
        until ( hdrtemp = nil )
      ; writestr ( '****leave procedure testwritefds****(:10:)' )
      end (*procedure testwritefds*)

(*************************************************************************)
(* this procedure builds and initializes the table to be manipulated     *)
(* by ullmans algorithm and returns the pointer to the head of this      *)
(* structure to the calling routine.                                     *)
(*************************************************************************)

  ; procedure buildtable
      ( var tblhead : tblptr
      ; attrct , relct : integer
      ; athead : attrptr
      )

    ; var rowindex : integer
      ; rowtemp : tblptr
      ; nextrow : tblptr

  (*************************************************************************)
  (* this procedure is local to procedure buildtable and simply builds     *)
  (* a row of the table and returns the pointer to the head of this        *)
  (* row to the calling routine.                                           *)
  (*************************************************************************)

    ; procedure buildrow
        ( var rowhead : tblptr ; rownumber , attrct : integer )

      ; var colindex : integer
        ; tptr : tblptr
        ; temp : tblptr

      ; begin
          new ( tptr )
        ; rowhead := tptr
        ; temp := tptr
        ; temp @ . attrcode := 1
        ; temp @ . row := rownumber
        ; temp @ . tag := rownumber
        ; for colindex := 2 to attrct
          do begin
            new ( tptr )
          ; temp @ . rowptr := tptr
          ; temp := tptr
          ; temp @ . attrcode := colindex
          ; temp @ . row := rownumber
          ; temp @ . tag := rownumber
```

```
        end
      ; temp @ . rowptr := nil
      end (*procedure buildrow*)

(*************************************************************************)
(* this procedure is local to procedure buildtable and simply links  *)
(* a newly built row to the table.                                   *)
(*************************************************************************)

  ; procedure linkrow
      ( oldrow , newrow : tblptr ; attrct : integer )

    ; var colindex : integer
      ; oldtemp : tblptr
      ; newtemp : tblptr

    ; begin
        oldtemp := oldrow
      ; newtemp := newrow
      ; for colindex := 1 to attrct
        do begin
          oldtemp @ . colptr := newtemp
        ; oldtemp := oldtemp @ . rowptr
        ; newtemp := newtemp @ . rowptr
        end
      end (*procedure linkrow*)

(*************************************************************************)
(* this procedure is local to procedure buildtable and it navigates  *)
(* both the table and the attribute structure and it sets the tag    *)
(* field to zero for each cell of the table corresponding to the     *)
(* intersection of an attribute and a relation.                      *)
(*************************************************************************)

  ; procedure settag ( tblhead : tblptr ; athead : attrptr )

    ; var tbltemp : tblptr
      ; colsave : tblptr
      ; attemp : attrptr
      ; reltemp : relptr

    ; begin
        tbltemp := tblhead
      ; colsave := tblhead
      ; attemp := athead
      ; while ( attemp <> nil )
        do begin
          reltemp := attemp @ . inrel
        ; repeat while ( reltemp @ . relnumber <> tbltemp @ . row )
            do tbltemp := tbltemp @ . colptr
          ; tbltemp @ . tag := 0
          ; reltemp := reltemp @ . nextrelnum
```

```
            until ( reltemp = nil )
          ; attemp := attemp @ . nextattr
          ; colsave := colsave @ . rowptr
          ; tbltemp := colsave
          end
        end (*procedure settag*)

    ; begin (*procedure buildtable*)
        buildrow ( rowtemp , 1 , attrct )
      ; tblhead := rowtemp
      ; for rowindex := 2 to relct
        do begin
          buildrow ( nextrow , rowindex , attrct )
        ; linkrow ( rowtemp , nextrow , attrct )
        ; rowtemp := nextrow
        end
      ; settag ( tblhead , athead )
      end (*procedure buildtable*)

(*************************************************************************)
(* this procedure writes the salient information stored in the data    *)
(* structure constructed by the procedure buildtable.  the information *)
(* is written in row-wise fashion.                                       *)
(*************************************************************************)

  ; procedure testwritetable ( tblhead : tblptr )

    ; var tbltemp : tblptr
      ; rowsave : tblptr

    ; begin
        writestr ( '****enter procedure testwritetable****(:10:)' )
      ; tbltemp := tblhead
      ; rowsave := tblhead
      ; repeat writestr ( '**new row**(:10:)' )
        ; repeat writeint ( tbltemp @ . attrcode )
          ; writestr ( '*is the attrcode*(:10:)' )
          ; writeint ( tbltemp @ . row )
          ; writestr ( '*is the row*(:10:)' )
          ; writeint ( tbltemp @ . tag )
          ; writestr ( '*is the tag*(:10:)' )
          ; tbltemp := tbltemp @ . rowptr
          until ( tbltemp = nil )
        ; rowsave := rowsave @ . colptr
        ; tbltemp := rowsave
        until ( rowsave = nil )
      ; writestr ( '****leave procedure testwritetable****(:10:)' )
      end (*procedure testwritetable*)
```

```
(*************************************************************************)
(* this procedure builds and initializes the search list to be used by *)
(* ullmans algorithm to identify and store equal tag fields, or tuples *)
(* of tag fields, from the table in preparation for a possible table   *)
(* update.                                                             *)
(*************************************************************************)

  ; procedure buildsearchlist
      ( var srchlsthead : srchptr ; relct : integer )

    ; var index : integer
      ; srchtemp1 : srchptr
      ; srchtemp2 : srchptr

    ; begin
        new ( srchtemp1 )
      ; srchlsthead := srchtemp1
      ; srchtemp1 @ . tag := 1
      ; srchtemp1 @ . nexttag := nil
      ; srchtemp1 @ . linked := false
      ; srchtemp1 @ . first := false
      ; for index := 2 to relct
        do begin
          new ( srchtemp2 )
        ; srchtemp1 @ . nextcell := srchtemp2
        ; srchtemp2 @ . tag := index
        ; srchtemp2 @ . nexttag := nil
        ; srchtemp2 @ . linked := false
        ; srchtemp2 @ . first := false
        ; srchtemp1 := srchtemp2
        end
      ; srchtemp2 @ . nextcell := nil
      end (*procedure buildsearchlist*)

(*************************************************************************)
(* this procedure writes the salient information stored in the data    *)
(* structure constructed by the procedure buildsearchlist.             *)
(*************************************************************************)

  ; procedure testwritesearchlist ( srchlsthead : srchptr )

    ; var index : integer
      ; srchtemp : srchptr

    ; begin
        writestr
          ( '****enter procedure testwritesearchlist****(:10:)' )
      ; srchtemp := srchlsthead
      ; repeat writeint ( srchtemp @ . tag )
        ; if not srchtemp @ . first
          then writestr ( '*first is false*(:10:)' )
          else writestr ( '*first is true*(:10:)' )
```

```
      ; if not srchtemp @ . linked
        then writestr ( '*linked is false*(:10:)' )
      ; srchtemp := srchtemp @ . nextcell
      until ( srchtemp = nil )
    ; writestr
        ( '****leave procedure testwritesearchlist****(:10:)' )
    end (*procedure testwritesearchlist*)

(*************************************************************************)
(* this procedure executes a complete pass of ullmans algorithm and      *)
(* makes appropriate modifications to the table.  if no changes are      *)
(* made to the table a boolean to this effect is returned to the main    *)
(* program.                                                              *)
(*************************************************************************)

  ; procedure ullman
      ( tablehead : tblptr
      ; fdlisthead : fdhptr
      ; searchhead : srchptr
      ; var changed : boolean
      )

    ; var coltemp : tblptr
      ; fdhtemp : fdhptr
      ; attrtemp : fdptr
      ; searchtemp : srchptr
      ; attrcode : integer
      ; lhs : boolean
      ; newlhs : boolean

  (*************************************************************************)
  (* this function is local to procedure ullman and when passed an         *)
  (* attribute code it returns a pointer to the column of the table        *)
  (* corresponding to this attribute code.                                 *)
  (*************************************************************************)

    ; function column ( attrcode : integer ; tablehead : tblptr )
      : tblptr

      ; var tabletemp : tblptr

      ; begin
          tabletemp := tablehead
        ; while ( tabletemp @ . attrcode <> attrcode )
          do tabletemp := tabletemp @ . rowptr
        ; column := tabletemp
        end (*function column*)
```

```
(*************************************************************************)
(* this procedure is local to procedure ullman and simply loads the      *)
(* tag fields from the appropriate column of the table into the re-      *)
(* spective tag fields of the search list.                               *)
(*************************************************************************)

  ; procedure loadtags ( column : tblptr ; searchhead : srchptr )

    ; var coltemp : tblptr
      ; searchtemp : srchptr

    ; begin
        coltemp := column
      ; searchtemp := searchhead
      ; repeat searchtemp @ . tag := coltemp @ . tag
        ; searchtemp := searchtemp @ . nextcell
        ; coltemp := coltemp @ . colptr
        until ( coltemp = nil )
      end (*procedure loadtags*)

(*************************************************************************)
(* this procedure is local to procedure ullman and it links search       *)
(* cells with equal tag fields and sets the boolean fields first and     *)
(* linked to true as appropriate.                                        *)
(*************************************************************************)

  ; procedure linktags ( searchhead : srchptr )

    ; var srchsave : srchptr
      ; srchtemp1 : srchptr
      ; srchtemp2 : srchptr
      ; tagsave : integer

    ; begin
        srchsave := searchhead
      ; while ( srchsave <> nil )
        do begin
          if not srchsave @ . linked
          then begin
            srchtemp1 := srchsave
          ; srchtemp2 := srchsave
          ; tagsave := srchsave @ . tag
          ; srchsave @ . first := true
          ; srchsave @ . linked := true
          ; while ( srchtemp1 @ . nextcell <> nil )
            do begin
              srchtemp1 := srchtemp1 @ . nextcell
            ; if srchtemp1 @ . tag = tagsave
              then begin
                srchtemp2 @ . nexttag := srchtemp1
              ; srchtemp2 := srchtemp1
              ; srchtemp1 @ . linked := true
```

```
            end
          end
        end
      ; srchsave := srchsave @ . nextcell
      end
    end (*procedure linktags*)

(*************************************************************************)
(* this procedure is local to procedure ullman and it relinks the    *)
(* tag fields in the event that a functional dependency has more     *)
(* than one attribute on its left hand side and it resets the bool-  *)
(* ean field first to true as appropriate.                           *)
(*************************************************************************)

  ; procedure relinktags ( searchhead : srchptr )

    ; var srchsave : srchptr
      ; srchsame : srchptr
      ; srchdiff : srchptr
      ; srchtemp : srchptr
      ; tagsave : integer
      ; newlist : boolean

    ; begin
        srchsave := searchhead
      ; while ( srchsave <> nil )
        do begin
          if srchsave @ . first
          then begin
            newlist := true
          ; srchsame := srchsave
          ; srchdiff := srchsave
          ; srchtemp := srchsave
          ; tagsave := srchsave @ . tag
          ; while ( srchtemp @ . nexttag <> nil )
            do begin
              srchtemp := srchtemp @ . nexttag
            ; if srchtemp @ . tag <> tagsave
              then begin
                if newlist
                then begin
                  srchdiff := srchtemp
                ; srchdiff @ . first := true
                ; newlist := false
                end
              ; srchsame @ . nexttag := srchtemp @ . nexttag
              ; if not srchtemp @ . first
                then srchdiff @ . nexttag := srchtemp
              ; srchdiff := srchtemp
              end
              else srchsame := srchtemp
            end
```

```
        ; if not newlist
          then srchdiff @ . nexttag := nil
        end
      ; srchsave := srchsave @ . nextcell
      end
    end (*procedure relinktags*)

(*************************************************************************)
(* this procedure is local to procedure ullman and it resets the tag *)
(* fields for a column of the table corresponding to an attribute on *)
(* the right hand side of a functional dependency.                   *)
(*************************************************************************)

  ; procedure resettags
      ( searchhead : srchptr ; var changed : boolean )

    ; var srchsave : srchptr
      ; srchtemp1 : srchptr
      ; srchtemp2 : srchptr
      ; lowtag : integer
      ; tagsave : integer

    ; begin
        srchsave := searchhead
      ; while ( srchsave <> nil )
        do begin
          if srchsave @ . first
          then begin
            srchtemp1 := srchsave
          ; lowtag := srchtemp1 @ . tag
          ; while ( srchtemp1 @ . nexttag <> nil )
            do begin
              srchtemp1 := srchtemp1 @ . nexttag
            ; if srchtemp1 @ . tag < lowtag
              then lowtag := srchtemp1 @ . tag
            end
          ; srchtemp2 := searchhead
          ; srchtemp1 := srchsave
          ; repeat tagsave := srchtemp1 @ . tag
            ; repeat if srchtemp2 @ . tag = tagsave
                then begin
                  if srchtemp2 @ . tag <> lowtag
                  then changed := true
                ; srchtemp2 @ . tag := lowtag
                end
              ; srchtemp2 := srchtemp2 @ . nextcell
              until ( srchtemp2 = nil )
            ; srchtemp2 := searchhead
            ; srchtemp1 := srchtemp1 @ . nexttag
            until ( srchtemp1 = nil )
          end
        ; srchsave := srchsave @ . nextcell
```

```
        end
      end (*procedure resettags*)

(*************************************************************************)
(* this procedure is local to procedure ullman and simply updates        *)
(* the tag fields in the appropriate column of the table with the        *)
(* tag fields of the search list.                                        *)
(*************************************************************************)

  ; procedure updatetable ( searchhead : srchptr ; column : tblptr )

    ; var coltemp : tblptr
      ; searchtemp : srchptr

    ; begin
        coltemp := column
      ; searchtemp := searchhead
      ; repeat coltemp @ . tag := searchtemp @ . tag
        ; searchtemp := searchtemp @ . nextcell
        ; coltemp := coltemp @ . colptr
        until ( coltemp = nil )
      end (*procedure updatetable*)

  ; begin (*procedure ullman*)
      changed := false
    ; fdhtemp := fdlisthead
    ; repeat attrtemp := fdhtemp @ . nextfdattr
      ; newlhs := true
      ; searchtemp := searchhead
      ; repeat searchtemp @ . first := false
        ; searchtemp @ . linked := false
        ; searchtemp @ . nexttag := nil
        ; searchtemp := searchtemp @ . nextcell
        until ( searchtemp = nil )
      ; repeat attrcode := attrtemp @ . attrcode
        ; lhs := attrtemp @ . lhs
        ; coltemp := column ( attrcode , tablehead )
        ; loadtags ( coltemp , searchhead )
        ; if newlhs
          then begin
            linktags ( searchhead )
          ; newlhs := false
          end
          else if lhs
            then relinktags ( searchhead )
            else begin
              resettags ( searchhead , changed )
            ; updatetable ( searchhead , coltemp )
            end
        ; attrtemp := attrtemp @ . nextfdattr
        until ( attrtemp = nil )
      ; fdhtemp := fdhtemp @ . nextfd
```

```
        until ( fdhtemp = nil )
      end (*procedure ullman*)

(*************************************************************************)
(* this procedure scans the table a row at a time and writes the mess- *)
(* age *******lossless******* if a row with all zero tags is found     *)
(* and ********************** otherwise.                                 *)
(*************************************************************************)

  ; procedure verdict ( tblhead : tblptr )

    ; var tbltemp : tblptr
      ; rowsave : tblptr
      ; lossless : boolean

    ; begin
        writestr ( '****enter procedure verdict****(:10:)' )
      ; tbltemp := tblhead
      ; rowsave := tblhead
      ; lossless := true
      ; repeat repeat if tbltemp @ . tag <> 0
            then lossless := false
          ; tbltemp := tbltemp @ . rowptr
          until ( tbltemp = nil )
        ; if lossless
          then writestr ( '*******lossless*******(:10:)' )
          else writestr ( '**********************(:10:)' )
        ; lossless := true
        ; rowsave := rowsave @ . colptr
        ; tbltemp := rowsave
        until ( rowsave = nil )
      ; writestr ( '****leave procedure verdict****(:10:)' )
      end (*procedure verdict*)

(*************************************************************************)
(* this procedure writes the tag fields of the final table in column-  *)
(* wise fashion along with the attribute associated with each column.  *)
(*************************************************************************)

  ; procedure writefinaltable
      ( tblhead : tblptr ; attrhead : attrptr )

    ; var tbltemp : tblptr
      ; colsave : tblptr
      ; atemp : attrptr
      ; index : integer

    ; begin
        writestr ( '****enter procedure writefinaltable****(:10:)' )
      ; tbltemp := tblhead
      ; colsave := tblhead
      ; atemp := attrhead
```

```
      ; repeat writestr ( '**new column**(:10:)' )
        ; writestr ( '*the attribute for this column is*(:10:)' )
        ; for index := 1 to maxattrlength
          do write ( atemp @ . attrname [ index ] )
        ; write ( nl )
        ; atemp := atemp @ . nextattr
        ; writestr ( '*the tags for this column are*(:10:)' )
        ; repeat writeint ( tbltemp @ . tag )
          ; tbltemp := tbltemp @ . colptr
          until ( tbltemp = nil )
        ; colsave := colsave @ . rowptr
        ; tbltemp := colsave
        until ( colsave = nil )
      ; writestr ( '****leave procedure writefinaltable****(:10:)' )
      end (*procedure writefinaltable*)

(*************************************************************************)
(*                                                                       *)
(*                          main program                                 *)
(*                                                                       *)
(*************************************************************************)

  ; begin
      ccfile1 := 513
    ; ccfile2 := 513
    ; bcfile1 := 1
    ; bcfile2 := 1
    ; readschema ( headattrlist , relcount , attrcount )
    ; testwriteattr ( headattrlist , relcount , attrcount )
    ; readfds ( headfdlist , headattrlist )
    ; testwritefds ( headfdlist )
    ; buildtable ( headtable , attrcount , relcount , headattrlist )
    ; testwritetable ( headtable )
    ; buildsearchlist ( headsearchlist , relcount )
    ; testwritesearchlist ( headsearchlist )
    ; repeat ullman
          ( headtable , headfdlist , headsearchlist , tablechanged )
      ; testwritetable ( headtable )
      until ( not tablechanged )
    ; verdict ( headtable )
    ; writefinaltable ( headtable , headattrlist )
    end.
```

REFERENCES

Bernstein, P. A. [1976]. "Synthesizing third normal form relations from functional dependencies", ACM Transactions on Database Systems 1:4, pp. 277-298.

Cardenas, A. F. [1979]. Data Base Management Systems, Allyn and Bacon, Boston, Massachusetts.

Codd, E. F. [1970]. "A relational model for large shared data banks", Comm. ACM 13:6, pp. 377-387.

Codd, E. F. [1972]. "Further normalization of the data base relational model", in Data Base Systems (R. Rustin, ed.) Prentice-Hall, Englewood Cliffs, New Jersey, pp. 33-64.

Date, C. J. [1981]. An Introduction to Database Systems (Third Edition), Addison-Wesley, Reading, Massachusetts.

Tsichritzis, D. C. and F. H. Lochovsky [1977]. Data Base Management Systems, Academic Press, New York.

Ullman, J. D. [1982]. Principles of Database Systems (Second Edition), Computer Science Press, Rockville, Maryland.

AN IMPLEMENTATION OF A LOSSLESS JOIN ALGORITHM

by

KARL RICHARD KLOSE

B.S., Bucknell University, 1958
M.S., University of Alabama, 1962
M.A., University of Alabama, 1967
Ph.D., University of Alabama, 1970

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1983

ABSTRACT

While the relational data model is an important advancement in the theory of database design, it is not without its problems. In an attempt to reduce data redundancy and update anomalies relational decomposition is employed. It is frequently necessary to recombine some of these decomposed schemes to answer a query and all too often information is lost when two schemes are rejoined. This loss of information is termed a "lossy" join whereas, if no information is lost the term "lossless" join is used. It is possible for a database administrator to test a particular decomposition of schemes during the design phase of a database implementation by submitting the decomposition of schemes to an implementation of a "lossless" join algorithm. This work describes an implementation of this algorithm written in Pascal for the PAS32 compiler on the Interdata 8/32 computer under UNIX v7 at the Department of Computer Science, Kansas State University.