PRODUCTION PROGRAM GENERATION IN A FLEXIBLE DATA DICTIONARY SYSTEM

Tore Risch

University of Uppsala, Sweden

Abstract

A data dictionary system with a query compiler is implemented in a symbol manipulation language, separate from the underlying database system. The query compiler (or program generator) generates COBOL programs for database access. These programs are optimized at generation time using information from the data dictionary. The implementation technique makes it possible to combine pilot implementation with production implementation of database application programs. Furthermore, an example is given of how the architecture of the system is convertible to different underlying database systems.

Key concepts: data dictionary, program generator, query language compilation, query language interpretation, non-procedural query language.

1. Introduction

I will discuss the architecture and other aspects of a data dictionary system with a query language handler. The basis for the paper is a working system called LIDAM (LISP Data Manager), which has been used since September 1977 at our computer center.

The system works with an existing database system, at present a relational database system called MIMER[1,3]. It is however, viewed by the operating system as a separate program. The contents of the data dictionary is represented as data structures (lists, trees, tables) in a high level symbol manipulation language (LISP).

An important feature of the system is a program generator (query language compiler) to which the user can specify database accesses in a high level query language. The system then automatically generates production programs in COBOL which efficiently perform the specified searches. The programs are optimized at generation time. There is a detailed description of the optimization algorithm in[18]. I will give a short summary of the optimization method used. The production programs are interactively used by end users, who do not have to master any query language or have detailed knowledge of the database contents. For detailed specification of the output layout, a report generator is integrated into the program generator, as well as conventional programming features such as assignment statements and conditional statements.

The system design has made it possible to combine query language compilation with interpretation. In this way pilot implementations can be automatically converted into production programs. Berild and Nachmens[4] have shown the need for pilot implementations when designing database application systems. The idea has been used for the extension of a query language handler for MIMER written in LISP[6]. This is discussed in section four.

Furthermore, the design of the system makes it little dependent on the underlying database system. Thus, an earlier version of the system[12] worked with IMS[8,11]. At the end of this paper I will give a summary of the experiences of the conversion from IMS to MIMER.

2. System overview

An important property of the LIDAM system is the separation of LIDAM from the underlying database system, and therefore of the LIDAM data dictionary from the database system's data dictionary. Another important decision was to implement the system and the data dictionary in the high level symbol manipulation language LISP. I will first give a summary of the motivations for the design decisions. The relation between LIDAM and the database system is shown in Figure 1. In section 2.2 I describe briefly the program modules available in LIDAM. There is a more detailed system overview in[17,19].

2.1 Motivations for the design

For the implementation I have used the INTERLISP system[24], which is a dialect of the programming language LISP. A number of properties of this system have simplified the implementation. An overview of these and other properties of LISP is given by Sandewall[22].

The logical data structures I needed, lists and trees, are well supported in LISP. These data structures are very useful for internal representation of program code. Since it is possible to associate procedures and data with each symbol, the manipulation of the internal representation is simplified.

Some properties of LISP together with the system architecture have made it possible to combine compilation of the query language with interpretation.

CH 1534-7/80/0000-0343$00.75 © 1980 IEEE

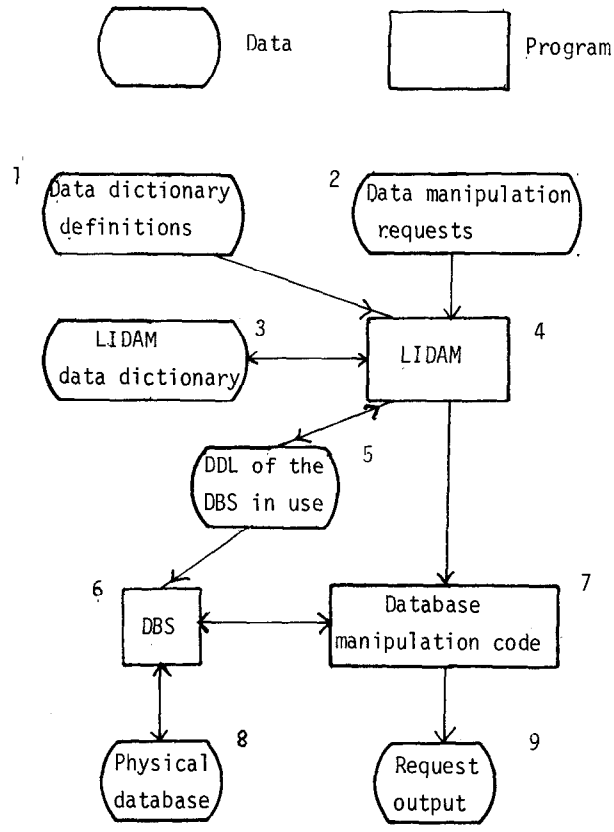

Figure 1.

The disadvantage of the programming system chosen is that such a flexible system as INTERLISP naturally will not have as good performance as conventional programming languages. However, I have not regarded the efficiency considerations as critical, since the system generates production programs (which are as efficient as possible of course) and this generation is made only a few times.

The reasons for separating the LIDAM data dictionary from the data dictionary of the underlying database system are in brief:

First, the system becomes less dependent on the underlying database system. Second, the data structures used in a detailed data dictionary has a complicated structure which is extensively manipulated. LISP is very convenient for representing such data structures. Since LISP is equipped with predefined I/O for its data structures, it is easy to make programs to save the internal list structure representation of the data dictionary on an external file and load it later.

The drawback of having the data dictionary separated from the database system is that the information stored in the data dictionary may become outdated as the physical database is modified. This problem is partially solved by programs which transform the contents of the data dictionary in the underlying database system into data structures in the LIDAM data dictionary and vice versa.

2.2 Program modules

The LIDAM top loop is the center of LIDAM. The different program modules in LIDAM are activated from this top loop by user commands.

Entry of the data dictionary: The data dictionary is normally created by a special data dictionary entry program. It is an interactive program that prompts the user for name, size, type, etc. of databases, fields and files. From the answers to these questions new parts of the data dictionary are created.

The DML program generator is described later.

The structure editor: A specially designed editor for the data dictionary is available. The editor checks that all changes are correct and admissible, in order to keep the data dictionary consistent.

LIDAM as documentation tool: An important use of LIDAM is as a documentation tool. The system can be used to answer queries about database items, e.g. which file descriptions are stored in the data dictionary, sizes of files and fields, relations between files, statistical values etc...

Views: LIDAM contains a module to define views (or external schemes). A view in LIDAM consists of a number of fields from some of the files in the database. From that view, the user may regard all of the database as a flat file with these fields. The program generator automatically maps the view onto the current database by using the data dictionary.

The view definitions may be stored symbolically in external files. When such a file is loaded the logical references in the view definition are connected to the corresponding physical description. Views may be defined in terms of other views. The design of the view feature makes view definition linkable to different LIDAM data dictionaries. In this way the same query can access different databases with similar content but different data structure.

3. Production program generation

I will give an overview of the program generation process in LIDAM, starting with a survey of the query language and the report generator. Finally, I will describe the internal functioning of the program generator.

3.1 The query language

The query language I use, LIDAM Request Language (LRL), is of a type similar to the relational database languages. In some respects, however, LRL is more powerful than many of the relational database languages available to-day. For instance, LRL allows multirelational queries, i.e. queries where the logical access paths are not specified by the user, but are determined automatically by the system. Similar techniques are used by Carlson and Kaplan[5], Osborn[16], Sagalowicz[21] among others. In certain other respects it is less powerful, as my intention has not been to construct a relationally complete query language[7] but to make a language which is user-oriented and solves practical problems. In LIDAM the database administrator regards the database as a network database, while the user has the relational view of the data.

A typical simple LRL statement has the form:

```
;RETRIEVE <output fields>  WHERE <predicate>;
```

For example:

```
;RETRIEVE DEPARTMENT,SALARY WHERE EMPLOYEE="SMITH";
```

The output fields (DEPARTMENT, SALARY) and the predicate (NAME="SMITH") may contain references to attributes in several different files (relations). The system automatically selects what intermediate files will be accessed and what access paths to be used.

In addition to query language constructs, LRL contains a programmable report generator, a simple dialogue generator to specify the form of the input to the programs generated, and conventional programming language constructs such as loop statements and procedures.

The users of the system have influenced the design of LRL, and particularly motivated the need for the report generator; the dialogue generator; multirelational queries; and views.

3.2 The report generator

For detailed specification of the output layout, there is a programmable report generator integrated into the query language. The reports are compiled into sections of the production programs from descriptions in high level report generator statements. By way of an example, in order to generate a program which repeatedly reads department names and prints the names and salaries of the employees of these departments, the statements will be:

```
 REPORT SALARIES(DEPT,EMP,SAL)=(
  CHANGE DEPT("SALARIES FOR DEPARTMENT";DEPT;//;
              "EMPLOYEE";20;"SALARY";//)
  EMP;20;SAL;
  SUMMARY DEPT(20;"-----";/;20;SUM(SAL)) );
REPEAT SALARIES RETRIEVE DEPARTMENT,EMPLOYEE,
    SALARY WHERE DEPARTMENT=PROMPT("INPUT
    DEPARTMENT NAME");
```

'20'means tabulation to position 20 and '/'means line feed. The specifications after 'CHANGE DEPT' are executed only when the value of DEPT is changed. In the same way, the specifications after 'SUMMARY DEPT'are executed at the end of a group of DEPT names. The other specifications are executed for each tuple retrieved.

```
  PROMPT("INPUT DEPARTMENT NAME")
```

generates code for reading the compare value from the terminal prompted by the specified prompt string. The formal parameters (DEPT,EMP,SAL) are bound to the output fields (DEPARTMENT,EMPLOYEE, SALARY) of the query to which the report is applied.

An example of an end user interaction with the program generated is:

```
INPUT DEPARTMENT NAME
?TOYS
SALARIES FOR DEPARTMENT TOYS

EMPLOYEE      SALARY

SMITH         1000
JONES         1750
BROWN          980
              - - - -
              3730

INPUT DEPARTMENT NAME
? etc.
```

Other features of the report generator are conventional programming language features such as assignment statements, arithmetic expressions and conditional expressions. The report generator can be used in a similar manner to the generators og RIGEL[20]. For example, a report to calculate and print the average value of some field can have the definition:

```
REPORT AVG(X)=(
       INIT(SUM:=0; CNT:=0)
       SUM:=SUM+X; CNT:=CNT+1;
       FINISH("THE AVERAGE OF";NAME(X);" IS";
           SUM/CNT))
```

The specifications after 'INIT'are executed initially before the retrieval, and the specifications after 'FINISH'are executed finally.

3.3 The program generator

LRL is a compiled language. The user gives a number of LRL statements to the LRL compiler, which are transformed into a COBOL program containing calls to the database system. The COBOL program is compiled by the COBOL compiler and executed the normal way.

A comparison is given in[17] of compilation versus interpretation of high level languages. The reasons why I have chosen to compile the query language are in brief: A more extensive optimization can be done, the generated programs will be of limited size, and the system will be less dependent on the underlying database system. In section four I will describe

how my design has made it possible to combine compilation and interpretation. Another example of a system having a compiled query language is System R[2,14]. Katz[13] has measured the considerable efficiency improvements for different levels of compilation of the query language in the INGRES system[23].

The method used encourages the specification of programs working over the database answering specialized queries by prompting the user for desired values of specific fields. The programs are very simple to use, and those who use them do not have to master any query language. It is my intention that the programs shall be used by casual users.

The original LRL statements are successively transformed by the program blocks in the program generator into new representations or data blocks. The program generator has three steps. They are illustrated by Figure 2, which illustrates the data flow from box one via box four to box seven in Figure 1.

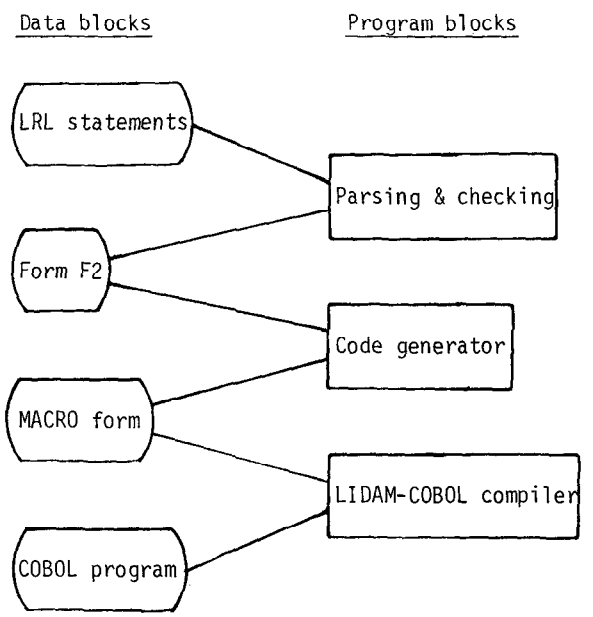

Figure 2.

Parsing and checking

The LRL statements are parsed in this program block and their syntactic and semantic correctness is checked. Incorrect statements must be rewritten. There is also a capability to correct some errors interactively when LIDAM finds them, and to do simple editing of the LRL statements. The output data block from this step represents the LRL statements parsed into an internal list structure form, form F2, where references to database items (files and fields) are replaced by pointers to the corresponding LIDAM descriptors. (A LIDAM descriptor is a data structure describing a database item.) The substitution of these pointers is done in parallel with a check on whether the items referenced exist in the data dictionary. This step also checks that the user has the authority to access the referenced database items, and the names of fields in the view are replaced by the corresponding field descriptors.

Form F2 is saved together with the original LRL statement. No further errors than those already detected by the checker can occur. Concomitantly, form F2 is guaranteed to be correct both syntactically and semantically.

The code generator

The form F2 data block is given as input to the code generator. A special user command collects all form F2 structures and gives them to the code generator. The form F2 structures are translated by the code generator into another data block, the MACRO form. This data block is a LISP oriented control structure describing database manipulations in the database system, and also describing other normal program operations (arithmetics etc). The structure of the MACRO form is thus independent of target language (COBOL at present) but contains special handles for the database system in use (at present MIMER).

The optimization method is applied in this step. It makes use of both indices and link tables of MIMER. Given a set of tuple identifiers (TIDs) for some database file, F1, the link tables are used to efficiently calculate theset of TIDs in another file, F2, participating in the equijoin[7] of the files F1 and F2 over a particular domain. This calculation is done without accessing any database records.

Haerder describes an impementation method to combine a similar link table feature with indices.[10]

The retrieval programs work in two phases, the collection phase and the distribution phase. Accesses to database records are avoided during the collection phase. In the ideal case, all accesses to database records are postponed to the distribution phase.

Among the files involved in the search, one file of particular importance is chosen, the FOCUS file. In the collection phase those TIDs of the FOCUS file are calculated which satisfy as large parts of the predicate (selection rule) as can be calculated by using indices and link tables. In the distribution phase this set of TIDs is used for accessing the records of the FOCUS file and the corresponding records in the other files from which data is to be retrieved. A form of tuple substitution is used[25].

Different selections of FOCUS file will result in different access times. Using a combination of analytic and heuristic methods, LIDAM selects the FOCUS file which seems to be the most promising. The amount of data to be accessed for different selections is estimated, using methods similar to those used in System R[9].

Some extensions of the method are made, handling cases where the preconditions for the method are

not ideal.

The LIDAM-COBOL compiler

The MACRO form is translated by a LIDAM-COBOL compiler into COBOL-source code. If other languages than COBOL are preferred (e.g.FORTRAN or assembler) this program module must be rewritten. The MACRO form is designed in such a way that it is simple to compile it into source code in any general purpose language.

The COBOL programs contain both calls to the database system and calls to a number of subroutines to conduct dialogues with the user and do report generation. Thus, it is assumed that there is a small runtime system for the generated programs.

Processing of generated programs

The generated COBOL code is written to a temporary file and the LIDAM system is exited. Then the generated program is completed with control commands. LIDAM generates control cards containing references to the OS-datasets where the physical database is stored and to the runtime system for LIDAM generated programs. At this point the generated program can be compiled and executed.

4. Combining query language compilation with interpretation

I will describe a method of combining compilation with interpretation of the query language which has been used in an implementation of a procedural query (and update) language for our database system (MIMER)[6].

An important difference between LISP and most other programming languages is that programs and data have the same representation. As a matter of fact, the programs are list structures of a particular form. These programs, represented by list structures, are interpreted by the LISP interpreter.

This property facilitates the writing of programs in LISP to manipulate other LISP programs. It is also easy to make programs generate other programs, and then immediately execute (interpret) the programs generated. This can be done without leaving the LISP system, unlike normal programming languages which have to be recompiled before execution.

There are a few other languages having this property, among them APL, SNOBOL and pure machine language. In APL and SNOBOL the programs are represented as strings instead of list structures. APL programming systems have also been constructed, even though they are not as advanced as the LISP programming systems. I know of no similar programming system in SNOBOL, although it is probably possible to construct one.

One interesting extension of the system is to make an interpreter in LISP for the MACRO form. When the MACRO form is generated, instead of translating it with the LIDAM-COBOL compiler (see Figure 2), it may be directly interpreted. It is possible to go even further; the different MACRO expressions may be defined as LISP functions. The normal LISP interpreter may then perform the interpretation.

To make it possible to interpret the MACRO form directly, handles must be built into LISP to access the database (i.e. the database system) . Thus, it must be possible to call the database access functions directly from LISP. Once the database system is accessible from LISP, the LRL compiler can be used both to generate specialized programs (in COBOL at present) and to generate and directly execute the MACRO form.

MIMER is a portable database system written in FORTRAN. In our department we have also developed a portable LISP system, LISP F3[15]. Since LISP F3 is written in FORTRAN it was relatively simple to make an interface berween LISP and MIMER by linking LISP F3 to MIMER and defining LISP functions corresponding to the database access routines of MIMER. This interface has then been used to implement a query language (MIMAN[6]) for MIMER, written in LISP. MIMAN has a syntax similar to QUEL[23]. At present no optimizer is included in MIMAN. MIMAN uses a technique to generate an executable LISP program which is directly interpreted.

Since MIMAN is written in LISP and has a similar design as parts of the program generator of LIDAM, the transformation of program modules from LIDAM to MIMAN is simplified. Thus, I have connected the report generator of LIDAM to MIMAN, making it possible to define a large class of application programs for MIMER directly in MIMAN. I have furthermore adapted the LIDAM-COBOL compiler for MIMAN.

Now MIMAN can be run in two phases:

1. During a pilot phase MIMAN is run interpretatively. The user may interactively write and test his application programs.

2. When the user is satisfied with the functioning of his application program, a command is given to generate the efficient production program in COBOL. The COBOL program is then used by the end user in a production phase.

By combining interpretation and compilation in this manner the program development is considerably simplified for programs definable in MIMAN. In addition it is simple to regenerate modified production programs previously generated. This improves the possibilities for the end user to influence the appearance of the programs.

5. Changing the underlying database system

I will give a summary of the experiences with the change-over of LIDAM from IMS (the database system used previously) to MIMER (the database system used at present). This is described in greater detail in my papers [17,19]. Three types of system changes were made:

First, old program modules were adapted to the new database system.

Second, the system was generalized in order to facilitate adaption to new types of database systems in the future. It should at least be adaptable on both IMS and MIMER. Since the system has been changed during the conversion, some work remains to extend LIDAM to work also with IMS.

Third, the system is extended with some wholly new facilities.

One reason for storing the access paths implicitly in the data directory (sec. 3.1) is to minimize the dependence on the underlying database system and on the database structure. My ambition has been to make it theoretically possible to use exactly the same LRL statement to specify a retrieval both for IMS and MIMER (and eventually also another database system). The use of views (sec. 2) makes it possible to have the same logical view of different databases that have the same contents.

The program generator is the module which is the most difficult to transform to work with different database systems. The parser and the checker remain about the same, while the MACRO form must be extended with new primitives for each new database system. The code generation for database independent parts of LRL may remain the same when changing database system. However, other code generation will differ considerably. The most difficult problem in the conversion of the program generator is the optimization algorithm. It is not only dependent on the overall structure of the database system but also on the detailed internal working of the database system.

6. Summary

I have presented a data dictionary system where the data dictionary is stored separately from the underlying database system, and it is represented as data structures in a high level symbol manipulation language (LISP).

The system can generate interactive production programs for end users from specifications in a high level query language. The production programs are optimized at generation time. The general principles of the query language, LRL, are discussed. Both the design and implementation are of interest. The query language allows a powerful type of queries, multi-relational queries, which makes it user-oriented and little dependent on the structure and type of the underlying database system. My practical experiences with LRL have shown the LRL-type of query language to be very useful for solving practical retrieval problems, even though LRL at present is not fully relationally complete[7]. Several LIDAM-generated programs are in practical use, and many of the features of LRL are developed from users' demands. The query language includes a report generator which is a very useful feature for practical production program specification.

The architecture of the query language compiler as well as properties of the programming language LISP has made it possible to use the query language both in compiling and interpreting mode. I have shown how this idea has been applied to the query language MIMAN[6] for our underlying database system.

The architecture of the system has made it possible to work with different underlying database systems. It also makes it possible to generate programs in one computer and execute them on other computers.

REFERENCES

1. S.Altemark, M.Jainz, S.Johansson, A.Persson, T.Risch and W.Schneider: "A Portable and User Oriented Database Management System and Its Applications in the Medical Field", 1980 Medinfo conf., Tokyo, Japan.

2. M.M.Astrahan, et. al.: "System R: Relational Approach to Database Management", ACM Transactions on Database Systems, June 1976.

3. A.Berghem, A.Haglund, S.G.Johansson, A.Persson: "A Partially Inverted Database System With a Relational Approach, MIMER (earlier RAPID)", Uppsala University Data Center, Uppsala, Sweden, 1977.

4. S.Berild, S.Nachmens: "CS4: A Tool for Database Design by Infological Simulation", Presented at VLDB 3, Tokyo, Japan, 1977
Published in "Tutorial: Software Methodology", IEEE Catalog No. EHO 142-0, IEEE Computer Society, 1978.

5. C.R.Carlson and R.S.Kaplan: "A Generalized Access Path Model and its Application to a Relational Data Base System", Proc. of the International Conference on Management of Data, Washington D.C., 1976, pp 143-154.

6. M.Carlsson: "MIMAN - a query language for DBMS Mimer", DLU 79/5, Datalogilaboratoriet, Sturegatan 2B, Uppsala, Sweden, 1979.

7. E.F.Codd: "Relational Completeness of Database Sublanguages", Data Base Systems, Courant Computer Science Symposium 6, Ed. R.Rustin, Prentice Hall, New York, 1972, pp 65-98.

8. C.J.Date: "An Introduction to Database Systems", Addison-Wesley Publishing Company, ISBN 0-201-14452-2, 1975.

9. P.Griffiths Selinger, M.M.Astrahan, D.D.Chamberlin, R.A.Lorie, T.G.Price: "Access Path Selection in a Relational Database Management System", ACM-SIGMOD 1979 Conf., Boston, Mass., 1979.

10. T.Haerder: "Implementing a Generalized Access Path Structure for a Relational Database System", Transactions on Database Systems (TODS), September 1978.

11. IBM, Information Management System Virtual Storage (IMS/VS) General Information Manual, GH20-1260-1

Information Management System/360, Version 2, Application Programming Reference Manual, SH20-0912-4

Information Management System/360, Version 2, Utilities Reference Manual, SH20-0915-2

12. M.Jainz, T.Risch(eds): "A Data Manager For the Health Information System Berlin", Computer Programs in Biomedicine 6, 1976.

13. R.H.Katz: "Performance enhancement for relational systems through query compilation", National Computer Conference, 1979.

14. R.A.Lorie, B.W.Wade: "The compilation of a Very High Data Language", IBM Research Report RJ2008(28098), May 1977.

15. M.Nordstrom: "LISP F3 User's Guide", DLU 78/4, Datalogilaboratoriet, Sturegatan 2B, Uppsala, Sweden, 1978.

16. S.L.Osborn: "Towards a Universal Relation Interface", Conference on Very Large Databases, Rio de Janeiro, 1979.

17. T.Risch: "Compilation of multiple file queries in a meta-database system", (Ph.D. Thesis), Dept. of mathematics, University of Linkoping, Linkoping, Sweden, 1978.

18. T.Risch: "Optimizing non-procedural multiple file queries", DLU 79/1, Datalogilaboratoriet, Sturegatan 2B, Uppsala, Sweden, 1979.

19. T.Risch: "A Flexible and External Data Dictionary System for Program Generation", International Conf. on Data Bases, Univ. of Aberdeen, Aberdeen, U.K. July 2-4, 1980.

20. L.A.Rowe, K.A.Shoens: "Data Abstraction and Updates in RIGEL", ACM-SIGMOD 1979 Conf., Boston, Mass., 1979.

21. D.Sagalowicz: "IDA: An Intelligent Data Access Program", Conference On Very Large Databases, Tokyo, oct. 1977

22. E.Sandewall: "Programming in an Interactive Environment: The LISP Experience", ACM Computing Surveys, Vol 10, No 1, 1978.

23. M.Stonebreaker, E.Wong, P.Kreps, G.Held: "The Design and Implementation of Ingres", ACM Transactions on Database Systems (TODS), Sept 1976, pp 189-222.

24. W.Teitelman: INTERLISP Reference Manual, XEROX Palo Alto Research Center, Palo Alto, Calif., 1974

25. E.Wong, K.Youssefi: "Decomposition - A Strategy for Query Processing", ACM Transactions on Database Systems (TODS), Sept 1976.