redit - an Editor for a Relational Database Environment

by

Robert John Wilson

A project submitted to

the Faculty of Graduate Studies and Research

in partial fulfilment of the requirements

for the degree of Master of Science (Applied)

School of Computer Science
McGill University
Montréal, Canada
March, 1987

To my dear wife Helen and daughter Ruth who put up with my absence during the period when this work was done

Acknowledgements

. . ., and without Him was not anything made that was made.

I am very grateful to Professor Merrett who encouraged me to come to McGill, taught me courses in relational databases, and was my supervisor for this project. His patience, criticisms, and direction have made this project a reality.

I am also grateful to Normand Laliberté who, permitted me to use functions that he had written as part of *relix* in this project.

I am finally grateful to all those who encouraged me in diverse ways during the time this project was undertaken, and during my stay at McGill University.

Table of Contents

1. Introduction

A relational database environment requires a set of such relational operations as projection, selection, join, edit and domain algebra to be applied on relations.

A relational editor performs the edit operation on a relation. Given a search key (made up of one or more attributes of the relation), the editor allows the user to manipulate the tuples in a relation through a template (form).

The concept of a software engineering programming environment calls for the creation of highly-interactive display-oriented tools for software development or data manipulation. Such an editor combines the functions of a normal text-editor with those of a parser or the semantics of the database environment, and allows a user to create and modify data entities (eg; tuples) in terms of a relational database environment, or programs in terms of a language syntactic structure.

This report describes the design and implementation of a relational editor — *redit*.

It provides a high-level interactive user interface. The basic operations to be performed are direct/sequential access, creation, insertion, deletion and updating.

Redit has two main modes and various submodes depending on the context of the operation being performed. The editor commands are therefore context-sensitive, highly-interactive, and are triggered by a single key stroke. Commands are used and named consistently so as to eliminate the need for memorizing them, and to reduce the time a user needs to become comfortable with *redit*. All the valid commands are displayed in a command window.

The editor has been designed to be compatible with the first release of the relational database implementation - *relix* running on VAX 11/780, Masscomps, and Cadmus Bip Plan running Unix Operating System. It can be ported to any Unix environment with virtually no modification.

The underlying principle in designing *redit* has been modularity, re-usability, and low degree of cohesiveness among the various modules. The program modules are thus easy to use or modify for a similar application.

Within the limitations of *redit* and *relix,* this design and implementation is in step with current research and development on the design and implementation of editors for a database or programming environment.

2. The Relational Database Environment

Present day database and information systems processing requires the transfer of tens of thousands of bytes of data from secondary to main memory at any instance. However, the digital computers that are used to process information are limited by the fact that data is transmitted from secondary storage to processor (main memory) in units of only one word the so called "Von Neumann bottleneck" [Back78]. The "Von Neumann architecture" is based on the design of ENIAC computers which began in 1943. These computers had limited processor memory as compared to secondary storage. As a result, programs and data were stored in secondary storage and transmitted one word at a time.

The need to cut down on the cost of processing large amounts of data in terms of time has motivated research in information processing.

The relational data base model for data representation was proposed by Codd [Codd70], as a rational for information representation. In relational databases, relations are used as a model for files and sets of relations as a model for data.

Various designs and implementations of Codd's ideas has resulted in such query languages as Structured English QUery, SEQUEL [Cham74], QUEL, which is based on relational calculus [Held75], Aldat, algebraic data language [Merr77], and many others. These languages allow a user to retrieve or modify the information in a database, knowledge databases and expert systems, and distributed databases.

A query language may be

- tuple oriented (tuple at a time) - relations are processed tuple by tuple.
- algebra oriented - operations are defined to take whole relations as operands and yield relations as the result.
- calculus oriented - use of an expression to describe the data to be retrieved; formulation of expression may be similar to first order predicate calculus.

An algebraic language considers a relation as the primitive unit of data , and thus provides a high level of abstraction. The closure property of algebraic operations ensures that the result of any algebraic operation is a relation. Aldat [Merr77] is an algebraic language proposed by Merrett in 1977, as a programming language for which relations are the elements. The Aldat project is a series of implementations of such a language on different computer systems at McGill University. These include :

MRDS - a data sub-language for PL/1 programming language [Merr76], which provided the user with project, select, and the complete array of set theoretic relational functions called μ-join.

MRDSP - a Pascal language implementation [Merr81] with the addition of σ-join operation, and extension of Codd's division [Codd79].

MRDSA - a UCSD Pascal language version [Chiu82], implemented as a data sub-language for Apple II microcomputer. This system provided the user with the extended set of relational operations, including project, μ-join, σ-join, a full screen relational editor and a QT-select function (an extension of the select operation) [Merr84] .

MRDS/FS - MRDS with functional syntax [Van83], used MRDSA on an IBM PC as a basis for interactive manipulation of relations.

All these implementations were relational database environments where Aldat was embedded in a programming language [Merr77].

Relix, the current implementation of Aldat, provides the user with a fast and easy to use query language [Norm86]. It provides the most complete and comprehensive implementation of Aldat in that it includes a picture editor and allows the concurrent editing of relations [Gunn87], a highly-interactive and context-sensitive relational editor, and can handle recursively defined relations [Norm86].

As a primary memory implementation of Aldat, the design of *relix* is based on the assumption that relations will be small enough so that the operands (at most two in any case), of any operation of the relational algebra can fit into memory.

This assumption places a limit on the size and number of relations; however, the trade off is a fast response time.

3. Issues in The Design of a Relational Database Editor

An editor is usually a user interface to any computer system environment, and they vary in complexity depending on its functionality within the environment. There are generally three classes of editors. These are :

- general text editors [Hans71] - for editing text files,
- graphics-based editors [Hert84] - graphical user interfaces,
- language-based program editors [Teit81] - a syntax-directed programming environment.

Language-based editors differ from general text and graphics-based editors, in that they incorporate the syntax of the programming language to help create syntactically correct programs by only permitting the entry of information that maintains a syntactically correct program. The user creates a new program in a top down fashion, generating program constructs and filling in the details. In many respects, it is similar to a fill-in-the-blanks style of editing, where the cursor can be moved to the desired position on the screen and entering the desired identifier or command [Kove86].

Such an editor may in addition have an incremental parser to further parse each statement into executable machine instruction as in the MUPE-2 system at McGill University [Madh84].

A relational database editor belongs to the class of language-based editors [Her84]. However, instead of being the interface to the environment from which other facilities or operations can be accessed, it is one of the operations defined in the relational database environment. A user may therefore bypass the relational editor, unless there is the need to use it.

The edit operation provides changes to the environment for either creating a new relation or re-designing (making changes to the attributes of a domain) an existing relation interactively.

In the design and implementation of *redit*, certain techniques were applied to improve the quality of the software and reduce maintenance and enhancement costs [Somm82]. These include basing the entire design on prototyping, which involves designing and testing the software incrementally as they are approved by the user.

This provides two advantages :

1. The design flaws were corrected and the implementation became simpler as more insight was gained into the functionality of the main parts of the editor.
2. The frequent interaction with the user ensured the software will meet the user's requirements when finally delivered.

In order to make the editor compatible with future enhancements and the frequent updates to *relix* during its design and implementation, a thorough analysis was made at the system, module, and code levels. At the system level, the need to design a consistent program structure independent of who applies it eliminated the need to for "quick and dirty" solutions which could lead to inconsistencies and finally to "quick fixes". A connection was established between the modules quite early in the design stage, in order to limit the degree of cohesion, and also to ensure low coupling between modules [Berg81], [Somm82].

At the module level, modules that are an integral part of others, shared common data with others; and the types of association (cohesion) among component elements within a module were identified. To limit coupling among the modules, hierarchical modularity (pure tree structures) was used in the structural design. This makes data abstraction, testing, and general program modification easier and simpler.

Extensive use was made of functions for data abstraction. A general dynamic data structure was designed for efficient use of memory, and was made sufficiently flexible enough for future extension to a more complex data structure.

Initial experiences in the design and implementation editors was gained in the design and implementation of SPED - a Structured Pascal EDitor [McGr85] for a Software Engineering

course (CS 308-762A), and DIRS - a Deductive Information Retrieval System [Wils86] for a course in Artificial Intelligence Methodologies and Programming(CS 308-763B).

4. Analysis and Design

4.1 Software Requirements Document

The relational editor is to provide a highly interactive and context-sensitive user interface to *relix*. The following are the basic functions of the editor :

i. To provide a mechanism for allowing the application programmer/user to view a relation algebraically as a set of tuples, or a relation tuple-at-a-time.

ii. To provide a consistent set of interactive commands to the user.

iii. To provide a command language to enable the user to perform the following operations :

- direct/sequential access of relations,
- creation of relations,
- insertion of tuples,
- deletion of tuples, and
- valid changes to tuples.

iv. To provide a mechanism for checking that

- domain values have the correct form;
- search keys and functional dependencies of the relation are not violated;
- integrity constraints on the relation are not violated.

4.1.2 Design Goals

In other to meet the software requirements specified in section 4.1, the following were established :

1. To make the editor highly interactive and easy to use, single key stroke commands shall be defined.
2. To implement context-sensitivity, functional areas shall be identified; and valid commands shall be defined for each function area.
3. For each major functional area, there will be a corresponding mode or submode in order to reduce the complexity of the design of the entire system.
4. Functions for data validation will be made general enough to be used by any functional area. This will ensure consistency in operations and also make it easier for data validation on attributes of the relation.
5. The User interface is to be organized in such a way that valid commands are :
    - displayed in a command window,
    - consistent within the editor, the database environment and Unix environment,
    - meaningful with regards to the function they perform.

In addition, screen windows will be designed so as to enable a user to distinguish between the main text (actual domain values), domain names, commands within each context, and error and informatory messages. Important, important items will be highlighted on the screen.

4.1.3 Constraints

The design constraints include :

i. The C programming language is being used for the ongoing development of *relix*. In addition, most of the implementation already done relie heavily on the Unix operating system and utilities. To make the editor compatible with the ongoing development and its future enhancements, the C programming language and Unix system utilities shall be used for the implementation of *redit*.

ii. The data structures to be used in the manipulation of tuples in *redit* has to be compatible with what has already been implemented for *relix*. Therefore to ensure compatibility, consistency and easy modification, certain functions have been imported from *relix*.

iii. The implementation is to be based on v2200+ and vt100+ terminals available at McGill university; however, it will be portable enough to operate on all terminals the can be connected to Unix operating system.

4.1.4 Processing

The editor will provide facilities for :

- creation, change and deletion of tuples in a relation,
- redesigning of an existing relation,
- access to all tuples in the relation either sequentially or directly, tuple-at-a-time.

The editor will operate on the following units :

Tuple - a record occurrence of a relation,

Search key - a minimal subset of the attributes of a relation that can be used to identify each tuple.

Template/Form - a structure for manipulating tuples. It will be made up of the relation name, domain name, and other relevant information that will be used to display the tuple on the screen.

4.1.3.1 Relation

A database is simply a collection of relations. Suppose SCHOOL is a database with COURSE_420 as a relation.

Figure 4.1 shows sample data for COURSE_420.

NAME	STUID	SEC	ASSIGN1	ASSIGN2	MID_TERM	FIN	AVG_MARKS	GRADE
amuesiwa, araba	8506273	2	8.5	9.0	18	48	90.1	A
berard, paulette	8314201	3	9.2	8.4	22	40	79.6	B
brady, vivian	8230267	1	4.4	6.8	16	44	71.2	B
christos, marilou	8215291	2	5.2	7.6	22	38	72.8	B
giroux, aline	8314626	1	8.0	6.4	24	46	84.4	B
halzeh, araba	8192214	1	7.2	8.0	18	42	75.2	B
hart, heather	8317112	1	4.8	4.4	16	25	50.2	D
jones, raymond	8215174	2	5.2	6.8	14	30	56.0	C
king, tam	8328521	3	6.8	8.8	24	36	75.6	B
lamontagne, paul	7913295	2	8.0	8.0	22	43	81.0	B
rivet, maurice	8214512	3	6.4	8.4	18	41	73.8	B

Figure 4.1 - an instance of an *m-ary* relation.

The form shown in figure 4.1 is an instance of an *m-ary* relation satisfying the following properties :

i. All rows are distinct.

ii. The ordering of the rows is immaterial.

iii. Each column is labelled making the ordering of columns insignificant.

iv. The value in each row under a given column is simple. For example, it does not have components such as (amuesiwa, araba; 7913295; 8.8; C) nor multiple values such as (8214512; 3).

The rows in figure 4.1 are called tuples or the "n-tuples" of the instance of a relation. An instance of a relation is therefore an abstraction of which figure 4.1 is only a representation.

The column headings (NAME, STUID, SEC, etc.) are the domains in the relation, and the

element in each domain is an attribute. A domain may thus have several or no attributes associated with it in different relations or in one relation.

A relation with the characteristics described above is called a *normalized relation* or a relation of the *first normal form.* A relational database is one which is made up of such relations. For a full treatment of relational database concepts refer to [Merr84].

4.1.3.2 Template/Form Design

A template/form is a representation for the display of relations on the screen. The template has the same name as the relation and is created in a database called TEMPLATE. The TEMPLATE database is created as a sub-directory for each database.To make a template accessible to the database environment, it is stored as a relation.

It is possible to have more than one template for the same relation. A template therefore has a name and a sequence number. In addition, for each domain in the relation, the name, the maximum possible length, and the row and column coordinates for displaying the domain are stored as attributes of the template relation.

Figure 4.2a is the template relation COURSE_420, the default template for COURSE_420.

The name "temp" in this instance was supplied by the user. Figure 5.3(ii) shows how the template looks on the screen.

TNAME	TDOM_NAME	TNUM	TLENGTH	TROW	TCOL
temp	NAME	01	26	00	00
temp	STUID	01	07	01	00
temp	SEC	01	02	02	00
temp	ASSIGN1	01	13	03	00
temp	ASSIGN2	01	13	04	00
temp	MID_TERM	01	11	05	00
temp	FIN	01	11	06	00
temp	AVG_MARKS	01	13	07	00
temp	GRADE	01	02	08	00

Figure 4.2a - template relation COURSE_420, a default template for COURSE_420.

A template may be manipulated to suit the user's needs. A separate functional area is created to allow the default template to be re-designed (to be moved to any position on the screen).

5. Implementation

5.1 Data Structures

A template/form is created for each relation, and is made up of the following data :

tname - template name (supplied by the user - default is relation name).

tdom_name - domain name in relation.

tnum - template sequence number.

tlength - length of a domain.

trow - row coordinate for displaying domain name and value.

tcol - column coordinate for displaying a domain name and value.

A simple linked list is used for storing a template for the session. This is to allow for easy manipulation, and also to ensure efficient use of memory by the general dynamic nature of linked lists. It also makes it easy for future generalization of the structure to form other complex combinations of data representation.

In this implementation, a linked list is basically an ordered sequence of elements (nodes). Such a list has a head (beginning), and a tail (end). Since lists are generally dynamic in nature (that is the number of nodes cannot be predicted before run-time), pointers are used instead of arrays to achieve efficient and reliable implementation. To allow for future modifications or reusability, a general list is designed in which each node of a list can contain several pieces of data. A structure of type *TEMPLATE_REL* corresponds to the data to be stored at each node.

TEMPLATE_REL contains the actual data (tname, tdom_name, tnum, tlength, trow, tcol) in the list structure and is illustrated in 5.1a. *TEMPLATE_REL* can thus be modified by either adding or deleting some of the data in the structure.

To make the structure more general, another type *TNODE* has been defined. This structure is made up of a pair of pointers.

The first points to the data associated with the node (*TEMPLATE_REL*), and the second gives the address where the next node can be found. Combining both *TNODE* and *TEMPLATE_REL* leads to the linked-list structure in figure 5.1b.

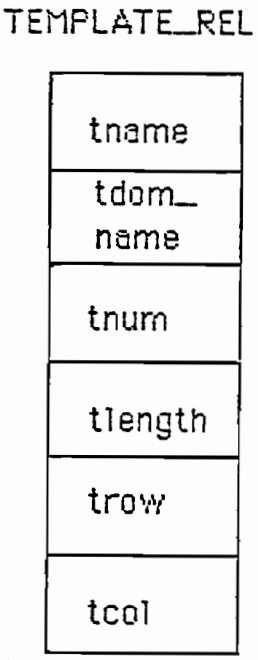


Figure 5.1a - illustration for type structure TEMPLATE_REL.

Finally, a list head which acts as an interface between the list and the functions that manipulate it is defined as a separate structure, and contains only the information required. This eliminates waste of memory space and also allows the addition of an extra information about the list; such as the current number of elements in the list and a pointer to the last element of the list. A structure type *T_HEAD* is defined and corresponds to the data to be stored in the header.

The type *T_HEAD* is illustrated in figure 5.1c.

The entire structure for representing a template is as shown in figure 5.1d.

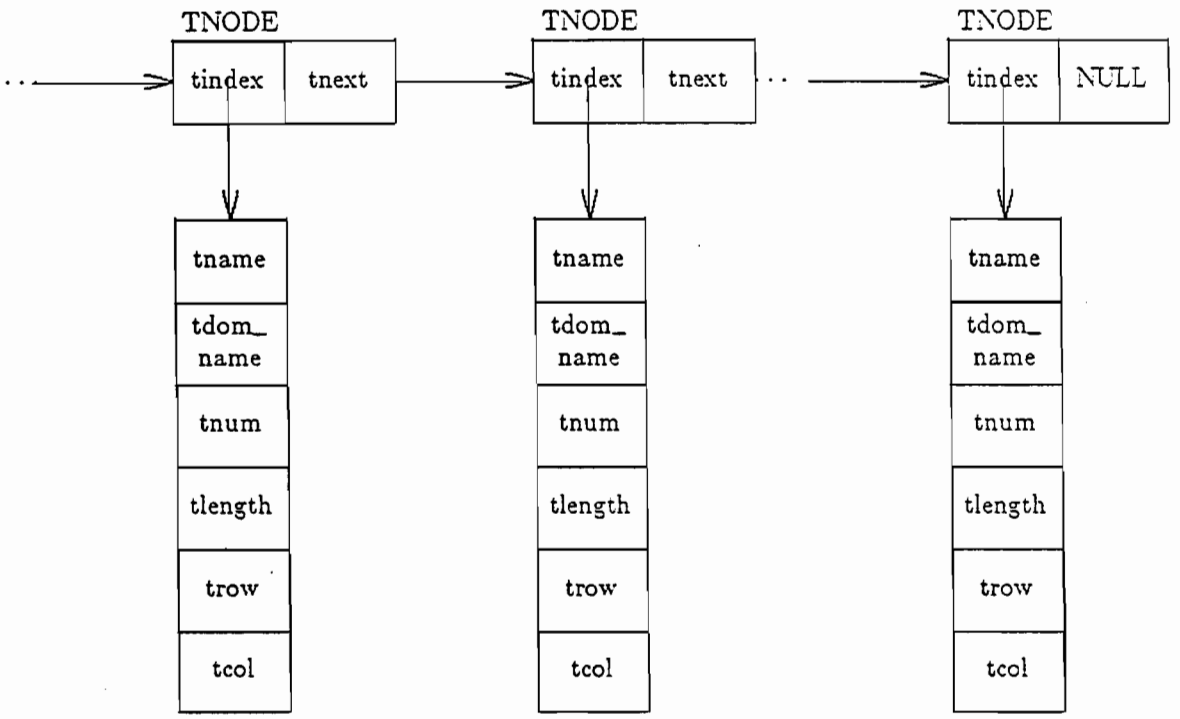


Figure 5.1b - illustration for type structure TNODE.

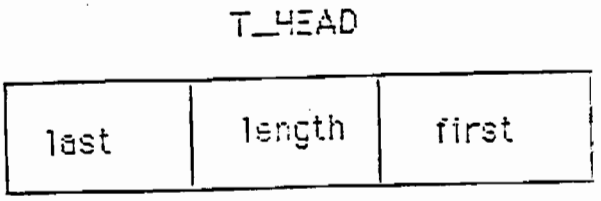


Figure 5.1c - illustration for type structure T_HEAD.

Functions for implementing special operations on the list have been implemented.

These include functions for creation of new lists (list head, nodes, and data), insertion into

the list, appending into the end of the list, and deletion of nodes in the list.

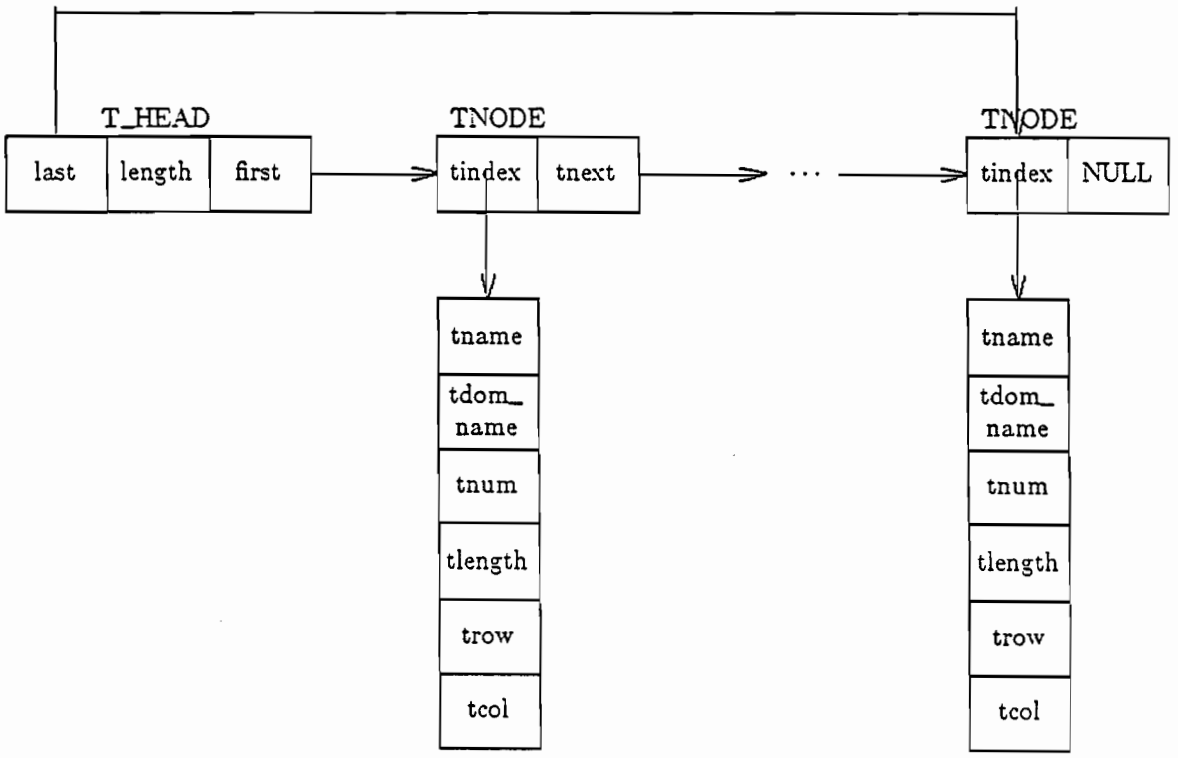


Figure 5.1d illustrating the representation of a template in memory.

5.2 Systems Design Diagram

The entire system was organized as a hierarchy of modules. This is to help reduce the complexity of the entire system; and also to maintain hierarchical modularity and thus low coupling among the program modules. Such a design ensures reusability of program modules and consistency in the general system structure.

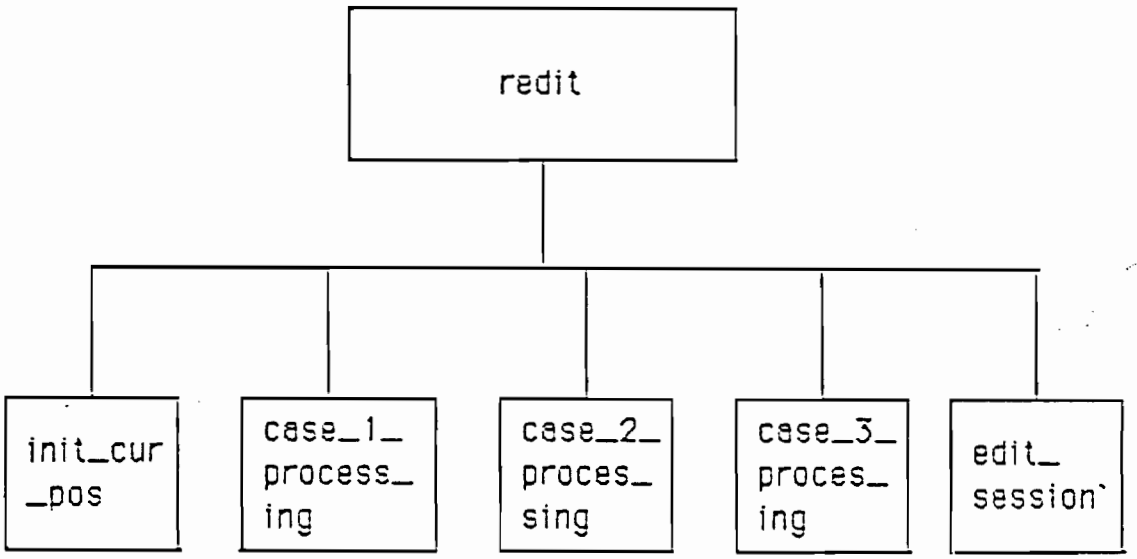


figure 5.2a - structure of *redit*

The main module (*redit*) interfaces with other modules in each function area as shown in figure 5.2a. It calls *init_cur_pos* to initialize terminal characteristics, window structures, and cursor variables. *Case_1_processing*, *case_2_processing*, and *case_3_processing* verifies the list of arguments supplied at the invocation of the editor, and sets the appropriate flags to enable the system to select the most appropriate mode as follows :

Name	Mode	Description
case_1_processing()	process	both relation and domain list are given.
case_2_processing()	process/design	only a relation is given.
case_3_processing()	design	only a domain list is given.

The system automatically switches to design mode if no template exits for the relation. Control is then passed to edit_session which merely switches the user to the relevant mode based on the various flags that has been set. Its structure is as shown in figure 5.2b.

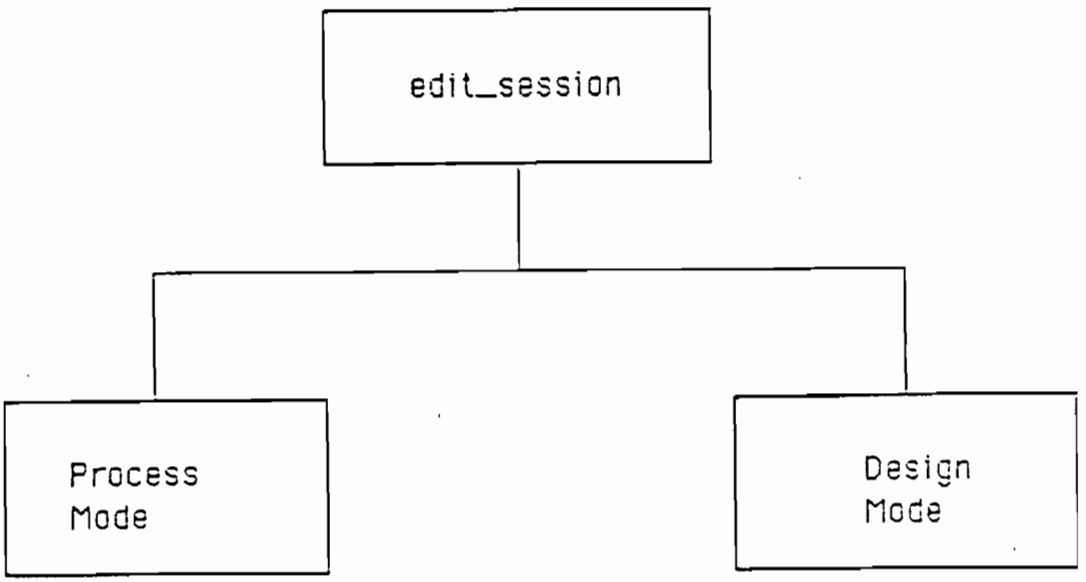


Figure 5.2b - structure of edit_session

The system switches to either process_mode or design_mode based on the flags that have been set as explained above. The session remains in a given mode until the user switches mode or decides to quit the editor. Edit_session therefore only controls the switching between the two top level modes, ensuring that the appropriate flags have been set; it also transfers control to the main module when the user decides to quit the editor.

Process_mode controls all low level processing of a relation. At the top level it only handles micro changes or commands. Changes involving tuples such as displaying next and previous tuples, deletion of tuples, and cursor movement on the screen are handled directly.

At the low level, macro changes are handled by a submode. Insertions, searching and changes to a domain are handled by submodes. These submodes handle processing of specific domains, and implement a specific function as a module. Figures 5.2c(i) and 5.2c(ii) depict the structure of these modules.

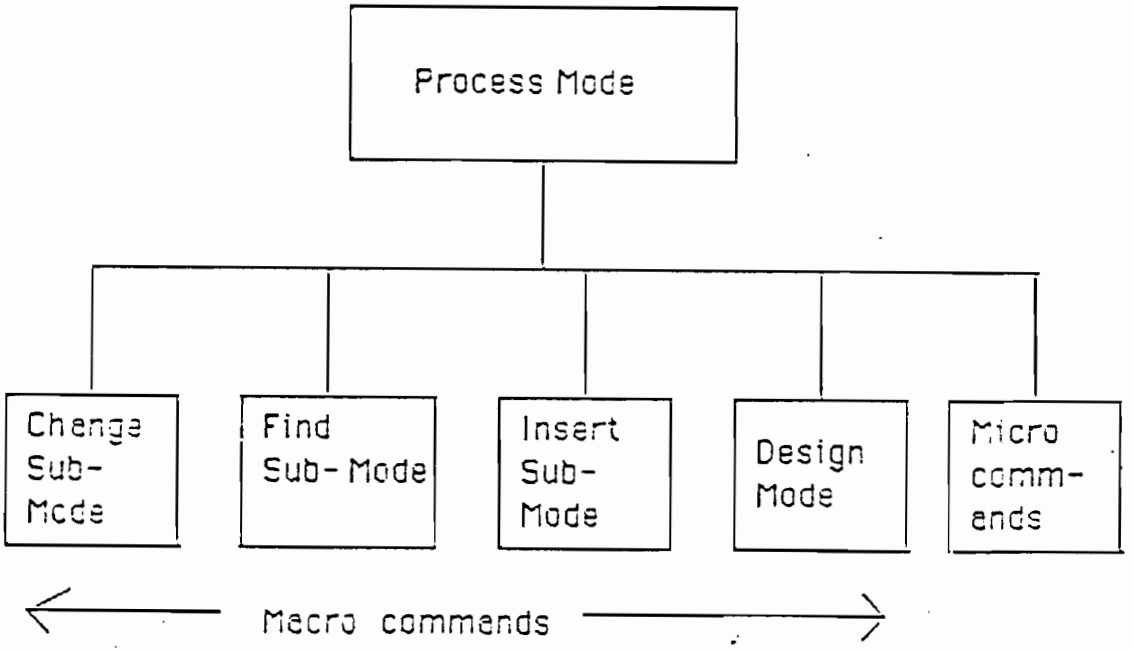


Figure 5.2c(i) - structure of process_mode

The Design_mode is responsible for manipulating templates. Its design is similar to Process Mode, and has Move and List submodes for implementing macro changes on the template. All other changes are handled directly at the top. This structure is shown in figure 5.2c(ii).

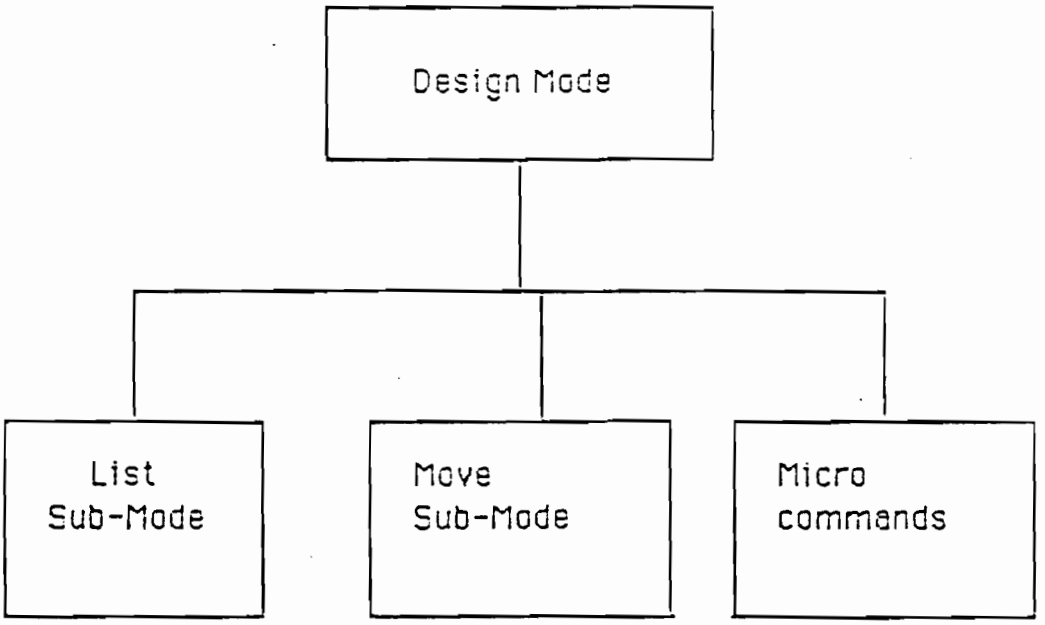


Figure 5.2c(ii) - structure of design_mode

5.3 User Interface

The editor has been designed for use on any standard terminal (particularly the v2200+ and vt100+ terminals available at McGill), that are compatible with Unix terminal handling protocols. Most of these terminals emulate standard screen size of 24 rows and 80 columns of display.

To help the user differentiate between error messages, valid commands and the main text, windowing facilities have been implemented using *curses* (a package of C language library routines).

The use of *curses* allows the system to do

- screen updating
- screen updating with optimization, and
- cursor motion optimization.

To update the screen optimally, the initial and final window characteristics are made global for accessibility. The screen is automatically refreshed to make it look neat each time a user enters a valid command or character (in the case of insertions). In addition the screen is redrawn automatically if there is a drastic change in the display. However, the user can redraw the screen whenever desired. The cursor motion has been optimized by using relative addressing for cursor positioning.

A data structure type called *WINDOW* in *curses* has been used to define two window structures. These are *relwin* and *cmdwin*. The window structures are based on the beginning row and column coordinates, number of rows, and number of columns.

Figure 5.3(i) and 5.3(ii) are pictorial representation of the screen in the Design and Process modes of *redit* using the relation COURSE_420 defined in figure 4.1.

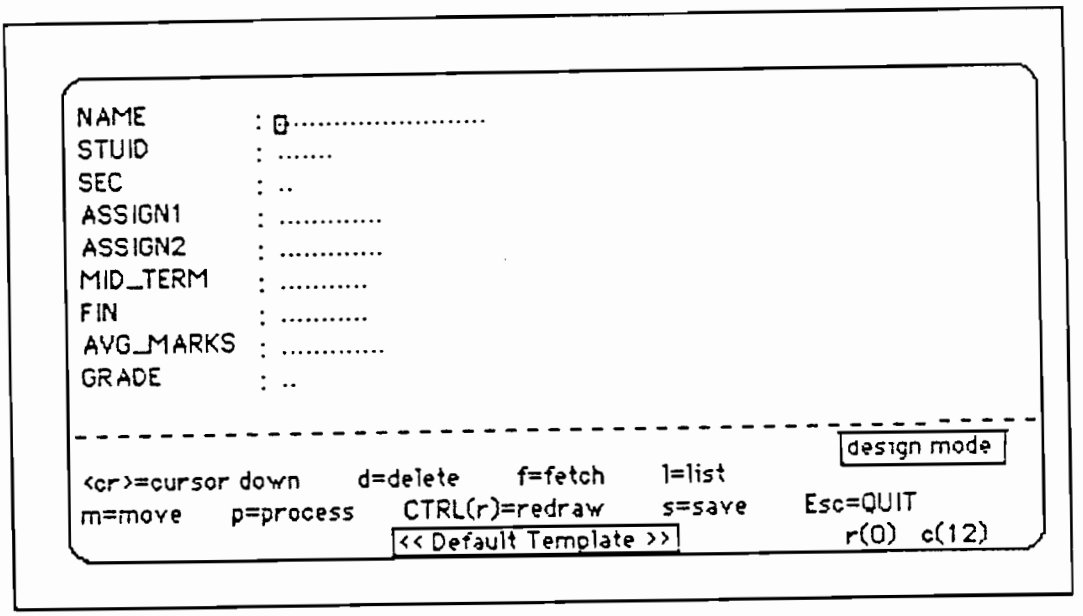


Figure 5.3(i) - pictorial representation of Design mode using the relation COURSE_420.

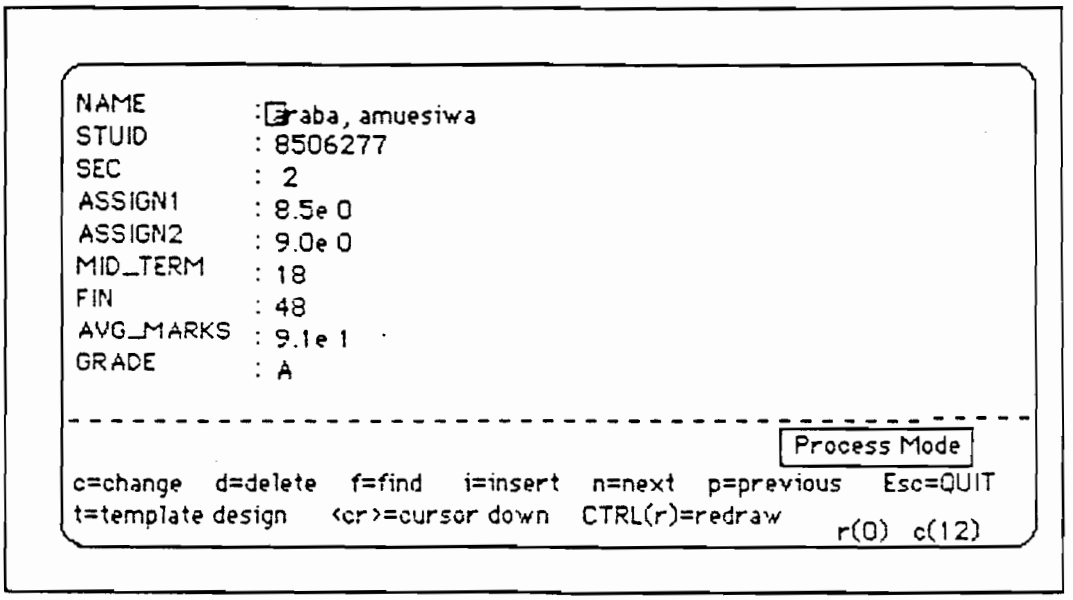


Figure 5.3(ii) - pictorial representation of Process mode using the relation COURSE_420.

6. Limitations and Extensions

The limitations of *redit* include :

1. The screen design is based on standard screen size (24 rows and 80 columns). However, for different screen sizes, only the portion corresponding to the standard screen size will be used. This may cause some display problems.
2. The size of the area for displaying a relation and its template is limited to 19 by 79 (rows 0 to 18; columns 0 to 78). This is because it has been assumed that no relation will have more than nineteen (19) domains. The need to have a sufficiently large command window reasonable enough to display the valid commands in each context (rows 19 to 24; columns 0 to 78) made this assumption necessary.
3. This implementation does not allow scrolling up or down.
4. Scrolling to the left or right is not not supported.

The extensions to *redit* in general can be considered as solutions to the limitations stated above. They are :

1. To extend the window to work properly on a non-standard screen, the size of "relwin" and "cmdwin" can be modified for the specific terminal by changing the initial value of *rel_NLINES* as explained in section 9.2.1.
2. Changing the argument to the function "scrollok" from "FALSE" to "TRUE" in the module "init_cur_pos" will implement scrolling. This will also provide a solution for the limitation specified in 3. above.

Although Unix does not support scrolling to either the left or right, it could be implemented by defining windows to contain the extension to the left or right of the screen. In this case a display of either window can be achieved by defining a command or a function key to be a call to a function to display the required window.

7. Conclusions

The design and implementation of *redit* has enriched the capability of *relix* by providing the edit operation or user interface for editing relations *relix* relational database environment.

Current software engineering techniques were applied in order to create an editor that can grow with future enhancements to the environment with little or no modification.
The use of prototyping technique provided the basis for a good design for the data structures, the modules, and the entire system as a whole resulting in a great reduction in its complexity.

As part of *relix*, it is consistent with its implementation and interacts properly with the entire environment. The use of "curses", and the importation of Unix terminal protocols makes it portable to any environment running Unix, especially on standard screens.

8. References

[Aho83] Aho A. V., Hopcroft J. E. and Ullman J. D., "Data Structures and Algorithms", Addison-Wesley, 1983.

[Back78] Backus J., "Can Programming be Liberated from the Von Neumann Style? A Functional and its Algebra of Programs", Comm ACM 21, 8, August 1978, pp. 613-641.

[Berg81] Bergland G. D., "A Guided tour of Program Design Techniques", Computer, Oct., 1981, pp. 13-37.

[Cham74] Chamberlin, D. D. and Boyce, R. F., "SEQUEL : A Structured English Query Language", SIGMOD, 1974, pp. 249-264.

[Chiu82] Chiu G., "MRDSA User's Manual" - Technical Report SOCS82.91, (May 1982).

[Codd70] Codd E. F., "A Relational Model of Data for Large Shared Data Banks", CACM, Vol. 3, No. 6, June 1970, pp. 377-387.

[Codd71] Codd E. F., "Relational Completeness of Data Base Sublanguages", in Data Base Systems (R. Rustin, ed.), pp. 65-98.

[Codd75] Codd E. F., "Understanding Relations", (ACM) FDT 7:3-4, 1975, pp. 23-28.

[Codd79] Codd E. F., "Extending the Database Model to Capture More Meaning", ACM TODS-4-4, Dec 1979, pp. 397-434.

[Gunn87] Gunnlaugsson, B. L., "Concurrency and Sharing for Prolog and a Picture Editor for Relix", McGill University, March 1987.

[Hans71] Hansen W. J., "User Engineering Principles for Interactive Systems", AFIPS, 1971, pp. 523-532.

[Held75] Held G. D., M. R. Stonebraker, and E. Wong, "INGRES - a Relational Database System", Porc. AFIPS NCC 44, Anaheim, May 1975., pp. 409-416.

[Hert84] Herot C. F., "Graphical User Interfaces, Human Factors and Interactive Computing Systems", 1984, pp. 83-103.

[John75] Johnson S. C. , "Yacc: Yet Another Compiler Compiler", Computing Science Technical Report No. 332, 1975, Bell Laboratories.

[Kame80] Kamel R. F., "The Information Processing Language Aldat: Design and Implementation", SOCS-80-14, August 1980.

[Kove86] Koved L. and Ben Schniederman, "Embedded Menus : selecting items in context", Commun. ACM 29, 4 (April 1986), 312-318.

[Lali86] Laliberté N., "Design and Implementation of a Primary Memory Version of ALDAT including Recursive Relations", M. Sc. Thesis, School of Computer Science, Mcgill University, August, 1986.

[Madh84] Madhavji N. H., D. Voulioúris, and N. Leoutsarakos, "The Importance of Context in an Integrated Programming Environments", Proc. 18th Annual Hawaii Int. Conf. of System Sciences, Hawaii, 1985 pp. 81-99.

[Merr76] Merrett T. H., "MRDS: An Algebraic Relational Database System", Canadian Computer Conference, May 1976, pp. 102-124.

[Merr77] Merrett T. H., "Relations as Programming Language Elements", Information Processing Letters, Vol. 6, No. 1, Feb. 1977, pp. 29-33.

[Merr81] Merrett T. H. and Zaidi S. H. K., "MRDSP User Manual", SOCS-81-27, August 1981.

[Merr84] Merrett T. H., Relational Information Systems, Reston Publishing 1984.

[McGr85] McGreggor A., and Wilson R. J. ,"SPED : A Structured Pascal EDitor", School of Computer Science, Mcgill University, Dec. 1985.

[Somm82] Sommerville I., "Software Engineering", Addison Wesley, (1982).

[Teit81] Tietelbaum T., and Reps T., "The Cornell Program Synthesizer: A Syntax-directed Programming Environment", Commun. ACM 24, 9(sept. 1981), 563-573.

[Wils86] Wilson R. J., DIRS - "A Deductive Information Retrieval System, School of Computer Science", McGill University, 1986.

[Vanr83] Van Rossum T., "Implementation of a Domain Algebra and a Functional Syntax", SOCS-83-18, August 1983.

9 Appendix

9.1 User Manual

9.1.1 Invoking Redit

Redit is invoked as follows :

R <-[*attribute list*] **redit A;**

where **R** is the resulting relation after the edit operation,

attribute list is any valid relational algebra expression, a partial or a complete attribute list of an existing relation.

A is either an existing relation or a valid relational expression.

The *attribute list* and **A** are optional, but at least one of them must be supplied. If **A** is omitted, then the user is prompted to enter at least one tuple before processing is done. In batch mode, the *attribute list* must be supplied.

The following are valid syntax for invoking redit.

R<-redit A;	-	uses the original sortlist (sequence) of the domain list of **A**.
R<-[*GRADE,NAME*] **redit A;**	-	the specified domain list is the new sortlist of **A**.
R<-[*NAME,STUID,GRADE*] **redit;**	-	creates a new relation **R** with the specified domain list. The user is prompted to insert at least one tuple before processing of tuples can begin.

NAME,STUID, and *GRADE* are domain list values of relation **A**.

9.1.2 Redit Command Set

Mode	Command	Explanation
Process	*c*	change - for modifying current tuple.
	<cr>	cursor down - moves cursor down by a row.
	d	delete - deletes current tuple.
	f	find - searches for a tuple, given one or more attributes.
	i	insert - appends tuple(s) to a relation.
	n	next - displays next tuple.
	p	previous - displays previous tuple.
	Esc	quit - quits the editor.
	CTRL(r)	redraw - redraws the screen.
	t	template design - switches to design mode.
Insert	*Esc*	abort & previous-level.
	c	accept & change & continue.
	<cr>	accept & continue.
	CTRL(a)	accept & previous-level.
Find	*Esc*	previous-level.
	n	retrieve next tuple.
	p	retrieve previous tuple.
Change	*a*	append - appends characters(s).
	x	delete char - deletes a character.
	y	delete toeoln - deletes a domain value.
	i	insert chars - inserts characters.
	Esc	previous-level - switches to previous level.
	r	replace chars - replace characters.
	u	undo domain - undo changes made to a domain value.
	U	undo tuple - undo changes made to a tuple.
	h	left - moves cursor to the left by a column.
	j	down - moves cursor down by a row.
	k	up - moves cursor up by one column.
	l	right - moves cursor to the right by a column.
Design	*<cr>*	cursor down - moves cursor down by a row.
	d	delete - deletes a template.
	f	fetch - fetches a template to be used for the session.
	l	list - display template.
	m	move - for modifying current template.
	p	process - switches to process mode.
	CTRL(r)	redraw - redraws the screen.
	s	save - save current template.
	Esc	quit - quits the editor.
Move	*h*	left - moves current structure left by a cloumn.
	j	down - moves current structure down by a row.
	k	up - moves current structure up by a row.
	l	right - moves current structure to the right by a column.
	CTRL(r)	redraw - redraws the screen.
	Esc	previous level.
List	*n*	next - displays next template.
	p	previous - displays previous template.
	Esc	previous - level

9.1.3 General Description of Redit Modes

Design and Process are the two main modes in which *redit* operates.

The Process mode sets the environment for processing tuples in a given relation; while the Design mode is for processing templates to be used in displaying tuples for the session. In all the modes the enter key (*<cr>*) provides a cyclic cursor movement by row. The system exits from either mode and returns to Relix when the *Esc* key is pressed.

A list of commands valid in each context are displayed in a command window, and a user need not memorize them. When Redit is invoked, i.e.,

R <-[*attribute list*] **redit A;**

the system enters either the Process or Design mode depending on conditions discussed below.

9.1.3.1 Process Mode

This screen is dispalyed if **A** is specified and it has at least one tuple. A corresponding template file (same name as **A**) must also exist; and contain at least one valid template for displaying **A**.

To modify tuples, the Process mode provides Change (*c*), and Insert (*i*) submodes. The Find (*f*) submode allows specific tuples to be retrieved from the relation. The system requests confirmation before the current tuple is deleted (*d*).

While in this mode if a different template is preferred, one can switch to Design mode (*t*); and either redesign the current one or retrieve a new one.

Additional set of commands are provided for displaying the next (*n*) and previous (*p*) tuples; and redrawing (*CTRL(r)*) the screen if it appears messy.

9.1.3.1.1 Insert Submode

In the insert submode, a new tuple can be appended to **A** by filling in the blank template with the desired value for each domain. The domain type and length are specified by the system, and also checks for valid characters for the domain. A tuple is rejected if it is a duplicate.

To abort and return to the previous level, *Esc* can be entered at anytime during the insertion. The backspace key can be used to correct typing mistakes within the same domain. If any previously filled domain is to be corrected, *Esc* can be entered and the whole operation can be restarted; or *c* can be used to switch to Change submode where the current tuple can be edited.

However *c* can be used only after the entire template has been filled.

The enter key ($<cr>$) can be used to continue the insertion of tuples. *CTRL(a)* will accept the current tuple and return to the previous level.

9.1.3.1.2 Find Submode

Specific tuples can be retrieved by specifying a unique domain value (the key) or one or more domain values of the tuple. The next (n) and previous (p) commands are useful when any other domain value(s) are entered and there are more than one tuple with the same value(s). To return to the previous level *Esc* has to be entered.

9.1.3.1.3 Change Submode

This mode allows the current tuple to be modified. The main commands are append (a), delete character (x), delete to-end-of-line (y), insert (i) and replace (r) characters. The cursor must be moved to the desired position before typing any of these commands. The cursor keys are up (k), down (j), left (h) and right (l).

To undo a change of a domain value use (u). *U* will undo all changes made to the current tuple.

9.1.3.2 Design Mode

The system enters the Design mode if **A** has no previous template or is omitted. The user is provided with a default template which can be modified for the session by going to the Move mode (*m*). Once the desired template has been created, the user can exit the Move mode (*Esc*), save the template (*s*) if he so wishes and proceed to the Process mode (*p*).

The delete (*d*) command can be used to delete any template, provided it exits and the name is specified correctly. If the current or session template is deleted, a default template will be provided automatically. To get information about templates, use list (*l*).

A previously saved template can be retrieved and made the session template by using the fetch (*f*) command.

The user is prompted to insert at least one tuple if there are no tuples to display before the system switches to Process mode.

9.1.3.2.1 Move Submode

In this mode a template can be redesigned by using the cursor keys up (*k*), down (*j*), left (*h*) and right (*l*); to physically move each template element to the desired position on the screen.

All operations involving the cursor keys are destructive, except the enter key (<*cr*>) which provides a cyclic cursor movement by row. *CTRL(r)* is provided for redrawing the screen.

9.1.3.2.2 List Submode

This is the basic command in the Design mode. It provides the means for finding out about template(s) available. The next (*n*) and previous (*p*) commands are useful if there are more than one template to display.

9.2.1 Window Structures

Two windows have been defined as follows :

name	from(row, col)	to(row, col)	no. of rows	no. of cols.
relwin	0,0	0,0	19	80
cmdwin	19,0	0,24	5	80

Relwin is used for displaying information about tuples and templates. In the Design mode, it is used to display template structures only; while the Process mode has both the session template and the current tuple displayed. The last row (row 18) is reserved for system messages pertaining to this window.

Cmdwin is provided for displaying the list of valid commands within the context of the type of processing being done. Run time information is displayed on row 24 (i.e. row 5 of *cmdwin*); and error messages are displayed on row 19 (i.e. row 0 of *cmdwin*).

The window definitions are based on *curses* routines available at any standard Unix environment. The *newwin* routine in *curses* was used to define the windows.

Thus with the declarations

```
WINDOW *relwin, *cmdwin;
int rel_row_begin, col_begin, cmd_row_begin, rel_NLINES, NCOLS,
  cmd_NLINES,
```

and initializations

```
NCOLS = 80;
rel_row_begin = col_begin = 0;
rel_NLINES = cmd_row_begin = 19;
cmd_NLINES = 5;
```

the definitions for relwin and cmdwin are as given below.

```
relwin = newwin(rel_NLINES, NCOLS, rel_row_begin, col_begin);
cmdwin = newwin(cmd_NLINES, NCOLS, cmd_row_begin, col_begin);
```

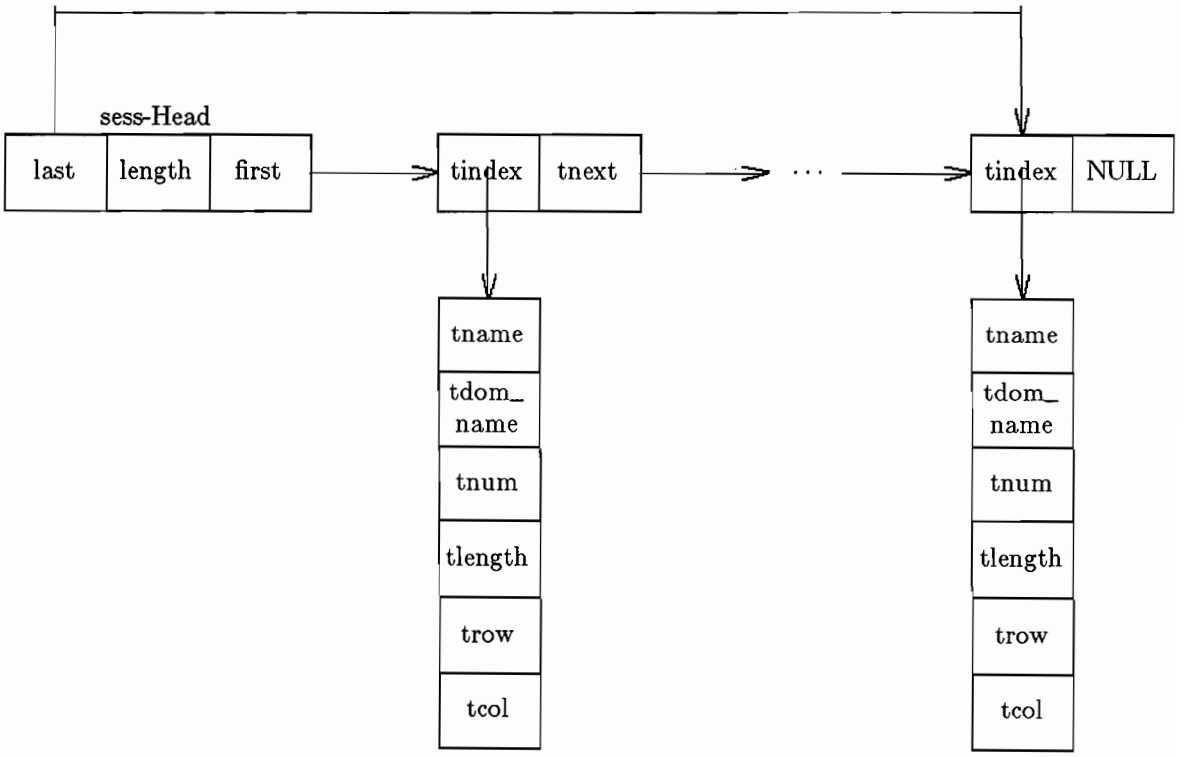
9.2.2 Template Structures

A template is represented as a relation with the following attributes :

tname	- template name (supplied by the user — default is relation name).
tdom_name	- domain name in relation.
tnum	- template sequence number.
tlength	- length of a domain.
trow	- row coordinate for displaying domain name and value.
tcol	- column cocordinates for displaying a domain name and value.

When Redit is invoked, the template information is stored as a linked list so as to ensure efficient usage of memory. The template information is either read from a file (if a template already exists) , or from the relation information stored in "rel-table" set up by Relix. Only one template is read into memory during processing.

The diagram of a link list below illustrates the representation of a template.



The list head is stored in a global variable *sess_Head*. Another global variable *TADDR* give the address of the selected template for the session. *Sess_Head* and *TADDR* have different addresses when the list command is executed in the Design mode. The declaration for the template list is as follows :

```
struct t_relation
        {
          char tname[ MAXTNAME ];
          char tdom_name[MAXTDOM];
          int tnum;
          int tlength;
          int trow;
          int tcol;
        } ;

typedef struct t_relation TEMPLATE_REL;

struct t_nodes
          {
            TEMPLATE_REL *tindex;
           struct t_nodes *tnext;
          } ;

typedef struct t_nodes TNODE;

struct t_header
          {
           int length;
           TNODE *first, *last;
          } ;

typedef struct t_header T_HEAD;
```

in addition to these declarations, if we let

*TNODE *tptr;*

tptr=sess_Head->first;

then;

tptr	accesses the first template relation structure.
tptr->tindex->tname	accesses the value in *name*
tptr->tindex->tdom_name	accesses the value in *dom_name*

. . .

tptr->tnext	accesses the next template relation value.

9.2.3 Module description

The module description below shows the general logic for the design of various modes of REDIT, and the interaction between them. Each module shows a list of functions that implement it. Some fuctions are written general enough to be used by more than one module. Although standard Unix functions and detailed description of each function has been omitted, they can be obtained by inspecting the source code.

The various modules are follows :

Name	Type	Description
redit()	int	main control module.
- *init_cur_pos()*	void	initialize windows and cursor coordinates.
- *case_1_processing()*	int	both relation and domain list supplied.
- *case_2_processing()*	int	only a relation is given.
- *case_3_processing()*	int	only a domain list is given.
- *edit_session()*	int	control module for various modes.
append_template_list()	int	append TEMPLATE_REL to list.
- *initialize_template_node*	TNODE	memory allocation & assignment.
binary_search()	int	binary search for specified domain values.
- *icrel_line()**	char	fetches a tuple from memory.
case_1_processing()	int	valid relation and domain list are given.
- *initialize_redit_env()*	TNODE	set up redit structures.
case_2_processing()	int	only a valid relation list is given.
- *initialize_redit_env()*	TNODE	set up redit structures.
case_3_processing()	int	only a valid domain list is given.
- *initialize_redit_env()*	TNODE	set up redit structures.
create_template_list()	T_HEAD	create template head.
- *T_MALLOC()*	int	allocate memory for T_HEAD.
change_win_context()	int	sets up commands for Change Submode.
- *icrel_line()**	char	fetches a tuple from memory.
- *bound_win()*	void	redraw window boundary.
- *display_tuple()*	void	display a tuple.
- *delete_toeoln()*	void	delete current domain.
- *ring_bell()*	void	beep signal.
- *error_msg()*	void	prompts user with error message.

* standard functions defined in RELIX environment.

delete_named_template()	void	delete specified template.
- *find_template()*	TNODE	display specified template.
- *mark_deleted()*	void	mark template for deletion.
- *display_default_template()*	void	display a default template.
- *ring_bell()*	void	beep signal.
delete_template_node()	TEMPLATE_REL	deletes a template node.
initialize_template_node	TNODE	allocation and memory assgnment.
- *T_MALLLOC()*	int	allocate memory for TNODE.
delete_tuple()	int	deletes current tuple.
- *bound_win()*	void	redraw window boundary.
- *icrel_line()**	char	fetches a tuple from memory.
- *sort_sess_tuples()*	void	sorts session tuples.
- *ring_bell()*	void	beep signal.
- *error_msg()*	void	prompts user with error message.
design_win_context()	int	sets up commands for Design Mode.
- *display_template()*	int	display a previously saved template.
- *display_default_template()*	void	display a default template.
- *bound_win()*	void	redraw window boundary.
- *move_emplate()*	void	modify template structure.
- *save_template()*	void	save current template.
- *delete_named_template()*	void	delete specified template.
- *fetch_template()*	void	retrieved specified template.
- *rel_create()*	int	create a new relation.
- *insert_win()*	void	initialize Insert Submode.
- *insert_win_context()*	int	sets up commands for Insert Mode.
- *redit_abort_msg()*	void	prompts user if there is no memory.
- *list_template()*	void	display all templates available.
- *row_col_pos()*	void	initialize cursor position in window.
- *error_msg()*	void	prompts user with error message.
- *ring_bell()*	void	beep signal.
display_tuple()	void	display a tuple.
- *get_shrink_list()**	int	fetches the absolute length of domain.
- *icrel_line()**	char	fetches a tuple from memory.
- *input_chars()*	void	set up for modifying a domain value.
dt_boolean()	char	accepts a valid boolean domain value.
- *dt_msg()*	void	prints valid commands in the context.
- *cur_pos()*	void	display cursor position in relwin window.
- *bound_win()*	void	redraw window boundary.
- *ring_bell()*	void	beep signal.
- *error_msg()*	void	prompts user with error message.
dt_integer()	char	accepts a valid integer domain value.
- *dt_msg()*	void	prints valid commands in the context.
- *cur_pos()*	void	display cursor position in relwin window.
- *bound_win()*	void	redraw window boundary.
- *FORMINT()**	void	format integer.
- *ring_bell()*	void	beep signal.
- *error_msg()*	void	prompts user with error message.

dt_real()	char	accepts a valid real domain value.
- *dt_msg()*	void	prints valid commands in the context.
- *cur_pos()*	void	display cursor position in relwin window.
- *bound_win()*	void	redraw window boundary.
- *FORMREAL()**	void	format real.
- *ring_bell()*	void	beep signal.
- *error_msg()*	void	prompts user with error message.
dt_string()	char	accepts a valid string domain value.
- *dt_msg()*	void	prints valid commands in the context.
- *cur_pos()*	void	display cursor position in relwin window.
- *bound_win()*	void	redraw window boundary.
- *error_msg()*	void	prompts user with error message.
edit_session()	void	control module for various modes.
- *process_win()*	void	initialize Process Mode.
- *process_template()*	void	displays template if it exists.
- *process_win_context()*	int	sets up commands for Process Mode.
- *design_win()*	void	initialize Design Mode.
- *process_template()*	void	displays template if it exists.
- *design_win_context()*	int	sets up commands for Design Mode.
error_msg()	void	prompts user with error message.
- *ring_bell()*	void	beep signal.
fetch_template()	void	retrieved specified template.
- *read_template_file()*	int	read template file if any.
- *find_template()*	TNODE	display specified template.
- *display_template()*	int	display a previously saved template.
- *display_default_template()*	void	display a default template.
- *ring_bell()*	void	beep signal.
find_tuple()	int	initialize Find Submode.
- *clear_tuple_area()*	void	clears the tuple on the screen.
- *tdots_tuple_display()*	void	display dots instead of tuple.
- *dt_boolean()*	char	accepts a valid boolean domain value.
- *dt_integer()*	char	accepts a valid integer domain value.
- *dt_real()*	char	accepts a valid real domain value.
- *dt_string()*	char	accepts a valid string domain value.
- *binary_search()*	int	binary search for specified domain values.
- *serial_search()*	int	serial search for specified domain values.
- *display_template()*	int	display a previously saved template.
- *display_default_template()*	void	display a default template.
- *display_tuple()*	void	display a tuple.
- *previous_tuple()*	void	displays previous tuple if any.
- *next_tuple()*	void	displays next tuple if any.
- *ring_bell()*	void	beep signal.
- *process_win()*	int	initialize Process Mode.
get_one_more_page()	void	allocate a page for insertion.
- *get_one_page()**	int	get one memory block.
- *enqueue_first_used()**	void	put memory block in used queue.
get_row_Map()	char	get characters on the screen.

initialize_redit_env()	void	set up redit structures for session.
- *make_sess_rel()*	void	make temporary relation for the session.
- *change_frozen()**	void	freeze the session relation.
- *icrel_get()**	int	get memory for the relation.
- *change_frozen()**	void	unfreeze the session relation.
- *icrel_fill()**	void	read session relation into memory.
- *sort()**	void	sort the tuples in the relation.
- *icrel_flush()**	void	write relation unto temporary relation.
- *assign()**	void	assign relation attributes.
- *icrel_free()**	void	free memory for relation.
- *change_frozen()**	void	freeze temporary relation.
- *icrel_get()**	int	get memory for temporary relation.
- *change_frozen()**	void	unfreeze temporary relation.
- *icrel_fill()**	void	read session relation into memory.
- *read_template_file()*	int	read template file if any.
- *select_template)*	TNODE	select template for session if any.
- *make_default_template()*	int	create a template for relation.
insert_template_list()	int	insert TEMPLATE_REL to list.
- *initialize_template_node*	TNODE	memory allocation & assignment.
insert_win_context()	int	sets up commands for Insert Mode.
- *change_sortlist()*	void	allocates memory for sorting tuples.
- *clear_tuple_area()*	void	clears the tuple on the screen.
- *tdots_tuple_display()*	void	display dots instead of tuple.
- *get_one_more_page()*	void	allocate a page for insertion.
- *dt_boolean()*	char	accepts a valid boolean domain value.
- *dt_integer()*	char	accepts a valid integer domain value.
- *dt_real()*	char	accepts a valid real domain value.
- *dt_string()*	char	accepts a valid string domain value.
- *icrel_line()**	char	fetches a tuple from memory.
- *change_win()*	void	initialize Change Submode.
- *display_tuple()*	void	display a tuple.
- *change_win_context()*	int	set up for Change Submode.
- *insert_win()*	void	initialize Insert Submode.
- *ring_bell()*	void	beep signal.
- *sort_sess_tuples()*	void	sorts session tuples.
list_template()	void	display all templates available.
- *bound_win()*	void	redraw window boundary.
- *display_template()*	int	display a previously saved template.
- *display_default_template()*	void	display a default template.
- *template_info()*	void	display template name and number.
- *error_msg()*	void	prompts user with error message.
- *ring_bell()*	void	beep signal.
list_template_structure()	void	list template - for debugging.
make_default_template()	int	create a template for relation.
- *create_template_list()*	T_HEAD	create template head.
- *T_MALLOC()*	int	allocate memory for TEMPLATE_REL.
- *insert_template_list()*	int	insert TEMPLATE_REL to list.
- *append_template_list()*	int	append TEMPLATE_REL to list.

make_template_file()	int	make default template from a file.
mark_deleted()	void	mark template for deletion.
- *read_template_file()*	int	read template file if any.
- *write_template_table()*	int	write template structure.
move_template()	void	Move Submode; modify template structure.
- *bound_win()*	void	redraw window boundary.
- *display_template()*	int	display a previously saved template.
- *display_default_template()*	void	display a default template.
- *get_blanks()*	int	get number of consecutive blanks on screen.
- *display_element_template()*	void	display template element.
- *invalid_pos_msg()*	void	gives messages about template position.
- *error_msg()*	void	prompts user with error message.
process_template()	void	displays template if it exists.
- *display_template()*	int	display a previously saved template.
- *display_default_template()*	void	display a default template.
process_win_context()	int	sets up commands for Process Mode.
- *display_tuple()*	void	display a tuple.
- *row_col_pos()*	void	initialize cursor position in window.
- *cur_pos()*	void	display cursor position in relwin window.
- *bound_win()*	void	redraw window boundary.
- *insert_win()*	void	initialize Insert Submode.
- *insert_win_context()*	int	sets up commands for Insert Mode.
- *redit_abort_msg()*	void	prompts user if there is no memory.
- *delete_tuple()*	void	deletes current tuple.
- *find_tuple()*	int	initialize Find Submode.
- *change_win()*	void	initialize Change Submode.
- *change_win_context()*	int	sets up commands for Change Submode.
- *sort_sess_tuples()*	void	sorts session tuples.
- *previous_tuple()*	void	displays previous tuple if any.
- *next_tuple()*	void	displays next tuple if any.
- *error_msg()*	void	prompts user with error message.
- *ring_bell()*	void	beep signal.
read_template_file()	int	read template file if any.
- *create_template_list()*	TNODE	create template head.
- *T_MALLOC()*	int	allocate memory for TEMPLATE_REL.
- *insert_template_list()*	int	insert TEMPLATE_REL to list.
- *append_template_list()*	int	append TEMPLATE_REL to list.
save_template()	void	save current template.
- *write_template_table()*	int	write template structure.
- *read_template_file()*	int	read template file if any.
serial_search()	int	serial search for specified domain values.
- *icrel_line()**	char	fetches a tuple from memory.
write_template_table()	int	write template structure.
- *int_to_string()**	void	converts integer to string.

9.2.4 Relix - an Implementation of a Relational Database Environment

The implementation of Relix is based on the relational database concept for data representation. It is interactive, in that it executes a single statement at a time. It is made up of two main modules; a parser generated by a Unix program called *yacc* [John 75] performs type checking on the statement and generates an intermediate code, and an interpreter which executes the intermediate code.

System Overview

The Relix internal representation is made up of a data dictionary and the internal representation of relations. A database name is a Unix directory name which is the same as the database name. The name must be unique and less than fourteen bytes. A database is thus a directory made up of two types of files. The first type is the file associated with a relation. For example a relation named *COURSE-MARKS* in a database *SCHOOL* corresponds to the file COURSE-MARKS in a directory SCHOOL. Relix does not recognize any form of abbreviation, and therefore the full name of a database or a relation must be specified.

The second type is made up of set of files that are used to perform housekeeping work on the database. These files are : TRACE, DOM, REL, RD, and all files with their names begginning with an underscore (eg; _NULL). TRACE contains a dump of all statements the user entered in addition to error messages generated by Relix; and can be used to provide a trace of a work session.

DOM, REL, and RD contains information used by Relix to maintain a data dictionary for the database. The information in these files are also stored as a relation. The relation DOM is defined on name, length and type. This relation is actually an extract of these attributes from DOM_TABLE, where all the information about domains are kept. All information about relations are kept in REL_TABLE. The relation REL is defined

on name, tuple_size, and ntuples is also an extract of these attributes from REL_TABLE. RD_TABLE contains the links between the domains and the relations in a database. RD is the contents of RD_TABLE, and the extract of the name attribute from REL_TABLE and domain name from DOM_TABLE. It is thus defined on relation name, domain name, count, dom_pos and sort_rank. A detailed description of Relix implementation is given in [Lalib86].