QUIN: Integration of Inferential Operators within a Relational Database

K. A. Spackman

MLI 83-31
ISG 83-13 / UIUCDCS-F-83-917

A publication of the *Machine Learning and Inference Laboratory*
Artificial Intelligence Center
George Mason University
Fairfax, Virginia 22030-4444 U.S.A.
(703) 764-6259

Editor: R. S. **Michalski**
Assistant Editors: J. Ralston and J. Zhang

The *MLI Reports* replace *ISG Reports*
published until December 1987
by the Artificial Intelligence Laboratory,
Department of Computer Science
University of Illinois,
Urbana, Illinois 61801

Quin:
Integration of Inferential Operators in a Relational Database

BY

Kent Alan Spackman

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

August, 1983

ISG 83-13

This research was supported in part by the National Science Foundation under Grant No. NSF MCS 82-05166 and in part by the Office of Naval Research under Grant No. N00014-83-G-0030.

QUIN: INTEGRATION OF INFERENTIAL OPERATORS IN A RELATIONAL DATABASE

BY

KENT ALAN SPACKMAN

B.Med.Sc., University of Alberta, 1977
M.D., University of Alberta, 1979

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1983

Urbana, Illinois

ACKNOWLEDGMENTS

Thanks go to Professor A. B. Baskin, my thesis advisor, for his support, patience and helpful suggestions during the development of QUIN, and to Professor R. S. Michalski for the ideas that prompted this project and his continuing contributions to its development. Earlier versions of the parser for the VL data language were programmed by Richard Schubert and Albert Cheng. The memory management routines for the ADVISE system were programmed by Heinrich Juhn, Mark Seyler and Lance Rodewald. The inference programs (CLUSTER, GEM, and others) were conceived by R. S. Michalski and were developed by many students under his supervision. Special thanks go to Robert E. Stepp for his work on CLUSTER, to Micheal Stauffer for his work on GEM, and to both of them for the development of the tabular input specifications for the inference programs.

I am indebted to the National Library of Medicine and the University of Illinois College of Medicine at Urbana for funding and the opportunity to pursue this work, and to Allan H. Levy and others at the Medical Computing Laboratory for administrative assistance, intellectual stimulation, and encouragement in my studies.

TABLE OF CONTENTS

CHAPTER 1

Introduction

This thesis describes a computer program called *QUIN* (*QU*ery and *IN*ference) which is a tool for database management and analysis. It represents a marriage between relational database and inductive inference technologies. Its purpose is the management of data and interaction with programs that use inductive inference to derive knowledge from examples. It has potential applicability in the logic-based analysis of data and in the creation of knowledge bases for expert systems. It can serve either as an independent data analysis system or as a module of the meta-expert system *ADVISE* [1].

1.1. Motivation

There are three problems which have motivated the construction of QUIN:

(1) Many scientific fields encounter problems with *conceptual data analysis*, finding patterns in data and providing conceptual or logical descriptions of such patterns. Conceptual data analysis is particularly applicable when data is discrete, unordered, nominal and incomplete, or when such data must be analyzed together with ordered continuous data. Although research data is usually analyzed using statistical techniques, inductive inference using logic-based formalisms can supplement traditional statistical methods in the analysis of databases. The field of medicine, for example, is one where there appear to be many problems that could be investigated with the help of a tool like QUIN.

(2) The traditional method of obtaining knowledge bases for expert problem-solving systems has been the difficult and time-consuming task of interviewing human experts and attempting to get them to codify their problem-solving knowledge and strategies. Inductive inference from examples can provide a mechanism for automatic generation of rules for such systems. Such

automated knowledge engineering should make the creation of expert systems faster and easier. It should also help to make the knowledge bases more consistent and complete.

(3) Manually manipulating large files of data and input parameters for several different inductive inference programs becomes a time-consuming and difficult task for users of such programs, particularly those users without extensive familiarity with computers. A convenient mechanism for managing data and for preparing the input to inference programs is needed. The need is more apparent when it is realized that the output of some programs is used as input to others, eventually leading to a feedback cycle involving several invocations of different programs.

1.2. Context

Recently, a research effort has begun to create an integrated system of software tools, known as ADVISE, for building and experimenting with expert systems. QUIN is an integral part of this system, although it may also be run independently. This section briefly describes ADVISE and the position of QUIN within it.

ADVISE is an expert system and also a tool for building expert systems. It provides expert problem solving and advice in chosen domains, such as plant pathology [2]. In addition, it creates a "workbench" environment for the knowledge engineer. In this environment he can create and modify the formal knowledge representations necessary to provide mechanized advice.

The following brief overview of the ADVISE system describes some of its unique features. Many expert systems have been and are being developed, some of which have succeeded in performing as well as any experts. Because there are so many expert systems it seems expedient to try to explain why ADVISE is not "just another expert system". It differs from most traditional expert systems in three important functions: knowledge representation, knowledge acquisition, and reasoning control.

(1) ADVISE provides three different forms of knowledge representation, allowing different knowledge to take different forms depending on what is most natural to the expert and the problem area. Condition-action rules are represented in a variable-valued logic rule formalism. Examples or tabular data are represented in a form natural to them, the relational table. Constraints or structural knowledge can be represented in a semantic network.

(2) Knowledge acquisition can take the form of traditional knowledge engineering practice which involves encoding and debugging knowledge formulated by a domain expert. In addition, ADVISE includes techniques for inductive inference which can be applied to generate knowledge from examples.

(3) There are multiple reasoning control mechanisms, each separated explicitly from the domain knowledge such that any available reasoning mechanism may be used to experiment and reason with a given knowledge base.

Figure 1 shows a diagram of the ADVISE program divisions and the position of QUIN within the system. QUIN provides the system with the means for relational representation of knowledge and with the mechanisms for invocation of inference programs for knowledge acquisition. The programs for inductive inference have been developed over the past few years at the University of Illinois under the supervision of professor R. S. Michalski [3].

1.3. Related Work

Many computer science researchers have recognized that investigators in many scientific fields need convenient mechanized tools for the management and analysis of data. The field of medicine has prompted many different systems that attempt to give clinicians ready access both to their data and to programs (usually statistical packages) that analyze the data. A few examples are CLINFO [4], RX [5], and MEDUS/A [6]. Unlike QUIN, these systems do not attempt to provide tools for the automated generation of rule-bases for expert systems.

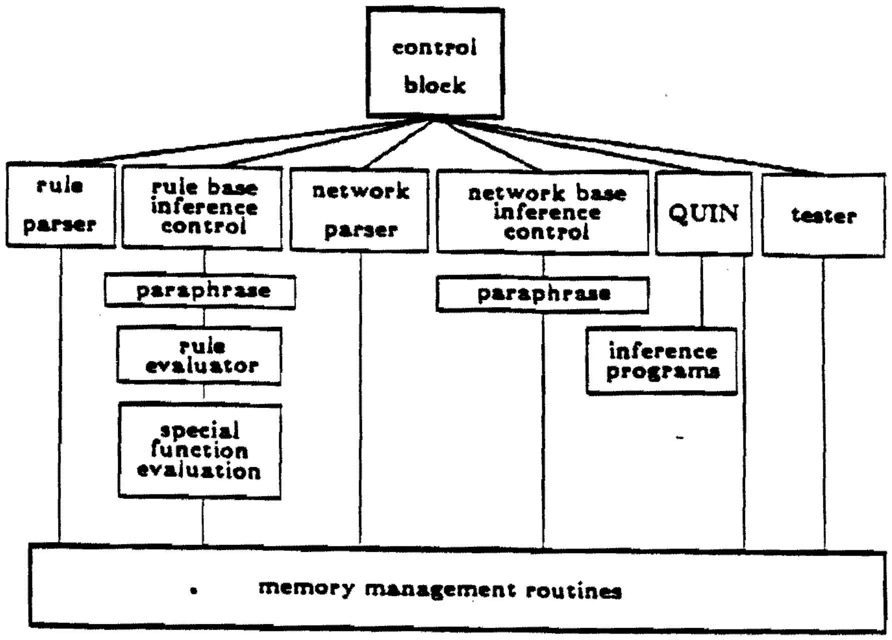
control block
rule parser
rule base inference control
network parser
network base inference control
QUIN
tester
paraphrase
paraphrase
rule evaluator
inference programs
special function evaluation
memory management routines

Figure 1. ADVISE Program Divisions

There has been work done to automate the process of rule acquisition from experts. The approach most often taken is to provide a convenient rule-editing mechanism that can be invoked at a point in the consultation where the expert finds the reasoning incorrect or the knowledge inadequate. One attempt at a solution to this problem was TEIRESIAS [7].

The acquisition of rules from an expert may be called "learning by being told," while induction of rules from examples may be called "learning from examples." The rules thus obtained are often called expert rules and machine rules, respectively. The relative performance of the two kinds of rules has been compared, and machine rules have been shown to perform as well or better than expert rules in one study from the field of plant pathology [8].

CHAPTER 2

Query and Inference

QUIN can be thought of as a tool for both the management and analysis of data. Management here refers to the organization, retrieval and modification of the data, while analysis refers to activities that attempt to discover more about 1) interrelationships within the data and 2) the phenomena that produced those interrelationships. Figure 2 illustrates the concept of QUIN as a dispatcher of a wide variety of operations that take relational tables as input and may return tables, rules or networks as output. These operations can be either data management (relational) operations or machine induction (inferential) operations. The databases that are used as input to inference programs can be conveniently handled with database management techniques that store, modify and restructure the data. The relational operations for this purpose are elaborated in chapter 3. The inference programs with which QUIN interacts form a set of inferential operations that are useful in sequence or in cycles with each other and the human critic. Details on the use of these operations are provided in chapter 5.

Thus, the tools provided by QUIN are:

(1) a convenient database management tool for reviewing and preparing examples for input to the inference programs and

(2) a uniform and user-friendly means for invocation of those programs.

A brief description of the implementation details of the program QUIN is contained in appendix A. The remainder of this chapter gives an overview of the concepts of database management (query) and analysis (inference) which are the fundamentals of the design of the system.

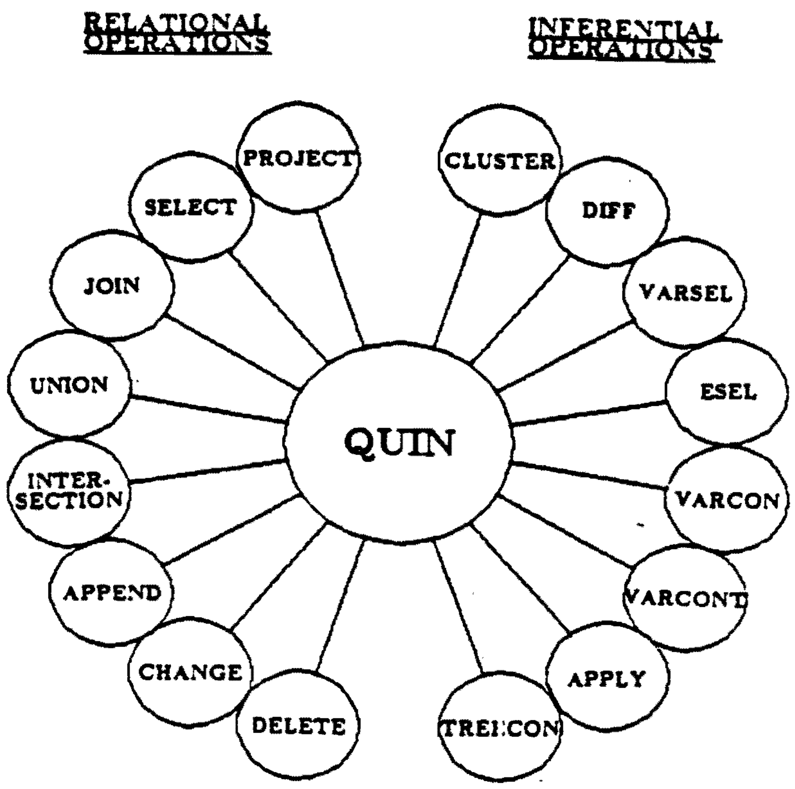

Figure 2. QUIN Operations

2.1. Database Management (QUery)

A database management system can be thought of simply as an electronic record-keeping system. Some purposes of database management are to facilitate the storage and retrieval of data by electronic means, to provide increased reliability, ease of access, decreased maintenance, reduction of redundancy and inconsistency, and maintenance of standards and integrity. The principles of this field are relatively new but have wide applicability. Database query operations provide decision-makers with convenient mechanisms to organize, select, format, and modify large amounts of information.

QUIN uses 1) a relational database model for management of data, combined with 2) a powerful query language based on the system of *variable-valued logic* [9].

- The relational model of data finds its origins in a paper by Codd [10] in the early 70's and is now considered one of three standard models for the organization of databases (the other two being hierarchical and network organization). See Date [11] for further discussion of these models. The interpretation and use of the relational model in QUIN is discussed in chapter 3.

- Variable-valued logic is a formalism for expressing logical statements. Because of its similarity to predicate calculus it can be used as a relational calculus for queries. The VL relational data sublanguage [12] was designed for this purpose and has similarities to Codd's ALPHA [13]. The syntax is also similar to the variable-valued logic syntax of the ADVISE rule base and the variable-valued logic syntax of the inference programs. It provides a succinct and easily learned language of interaction with QUIN's database. The details of the language are provided in chapter 4, and a grammar is included in appendix B.

2.2. Database Analysis (INference)

As stated previously, QUIN uses inference programs to perform conceptual data analysis. Describing data in terms of logical, functional and causal relationships is the goal of such analysis. It is accomplished by automated generalization of many examples to find *common* features of similar objects and *distinct* features of dissimilar objects. Descriptions of groups of objects and the important differences between them are also produced. Other objectives of conceptual data analysis include:

- finding the most efficient means of distinguishing between one group of objects and another,
- determining which attributes are most relevant in describing or differentiating groups,
- determining which objects are most representative of a group (typical or classical cases), and
- determining how attributes can be combined to provide new attributes which describe or differentiate better (such as ratios or differences).

Operation	Abbreviation	Program Invoked
clustering	cluster	CLUSTER/paf [14]
differentiation	diff	GEM [15]
variable selection	varsel	PROMISE [16]
event selection	esel	ESEL/2 [17]
variable construction	varcon	NEWVAR [18]
variable construction, time-oriented	varcont	CONVART [19]
application of rules to test cases	apply	AQ11 [20]
decision tree construction	treecon	OPTREE [21]

Table 1. Inferential Operations

Table 1 shows the meanings of the inferential operators and the names of the programs which they invoke. The output of one inference program may be used as input to others, permitting sequences of inference instructions to be issued followed by the evaluation of the results, possibly prompting further inference on the previous results. Cyclic processing of the data in this manner may eventually result in what may be called *knowledge refinement*, as illustrated in figure 3.

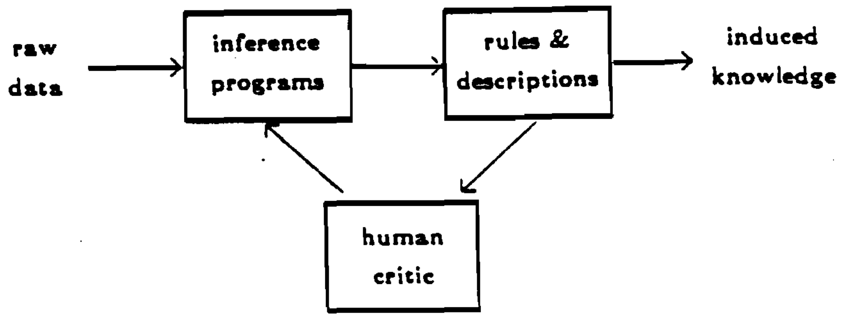

Figure 3. Knowledge Refinement Cycle

CHAPTER 3

The Relational Model

This chapter gives a brief overview of the relational model of database organization and describes the interpretation of the model by QUIN. The concept of a table of data and the way it represents the mathematical notion of a *relation* is fundamental to the relational model of data used by QUIN. The model also includes the concepts of *keys*, *normalization*, and *relational operations*, each of which will be discussed in turn.

3.1. Relational Tables

A **relational table** is simply a table that represents a relation. Tables are familiar as a format for representing data. Consider table 2, an example of a table of clinical laboratory values obtained from blood specimens.

labvals			
spec#	Hgb	MCV	RBC_morph
1024	10.3	78	microcytosis
891	13.1	90	normal
555	14.2	88	poikilocytosis
423	16.5	85	anisocytosis
425	11.1	78	microcytosis
455	10.4	77	microcytosis

Table 2. Clinical Laboratory Values

In table 2 each column corresponds to an attribute and each row represents an individual data object. The values within each row of the table represent the description of an object with respect to each of the attributes. Thus *spec#* refers to the specimen number, while *Hgb*, *MCV* and *RBC_morph* refer to the hemoglobin, mean corpuscular volume and red cell morphology of the specimen. These four names (spec#, Hgb, MCV and RBC_morph) comprise the *attribute list* of the table. The values obtained from each specimen occupy a single row in the table.

A **relation** is a set of ordered rows each of length n, (called *n-tuples*), where the value of the i^{th} column in a tuple (V_i) is drawn from a domain D_i. The relation has *domain sets* $D_1, D_2, ..., D_n$, where n is the *degree* of the relation. Table 2 is of degree four. Its domain sets include the set of all possible specimen numbers, the set of all possible Hgb values, the set of all possible MCV values and the set of all RBC morphologies. These domain sets need not be explicitly delineated in a database, but are important in the mathematical definition of the concept of a relation. For further reading see [11].

Relations are intuitively well represented as tables, but relational tables in QUIN differ in some ways from the strict interpretation of a mathematical relation. First, the attributes (columns) are named, and therefore two tables in which the only difference is altered column order are considered to be equivalent. Second, in relations the rows are not considered to be ordered, but QUIN allows rows to be ordered according to the values of attributes, e.g. in increasing order by index number (value-controlled ordering). Third, the "zeroth" row of a table in QUIN is occupied by the attribute list, and data then follows beginning with the next row.

3.2. Keys

A **key** is an attribute or combination of attributes that have unique values for each tuple in the relation. In other words, no two tuples in a relation may have identical values of the key attributes. This constraint ensures against duplication of data records. Some examples of keys include an identification number (such as specimen number in table 2), a unique name, or a unique combination of two more attributes, such as name and date. To allow purposeful duplication of data for use in the inference programs, a table may optionally have no key defined.

3.3. Normalization

A table is said to be *normalized* (in first normal form) if each entry in the table is non-decomposable, i.e. a table or set of values cannot constitute an entry in a normalized table.

Several levels of normalization have been defined (1st, 2nd, 3rd, Boyce/Codd, 4th, Projection/Join - see [11]) but the attainment and management of normalization beyond first normal form in QUIN is left entirely to the discretion and effort of the user.

3.4. Relational Operations

The relational model includes operations that take relations as input operands and give a relation as output. These operations can be classified as traditional set operations (union, intersection, difference and cartesian product) and special relational operations (project, select and join). These relational operations are incorporated within the query language provided in QUIN. They are briefly introduced here and examples of their implementation are given in the next chapter.

Union

The union of two relations is the set of all tuples contained in both relations (without duplication). To perform the union of two relational tables in QUIN, they must be *union compatible*, which means they must have identical attribute lists. The same constraint applies to the operations of intersection and difference.

Intersection

Intersection comprises the set of tuples common to both relations.

Difference

The difference of two relations is the set of tuples contained in the first relation but not in the second.

Product

The cartesian product of two relations is the set made up of the concatenation of each of the tuples in the first relation with each of the tuples in the second.

Selection

Selection provides a subset of tuples from a relation that meet certain selecting criteria. It produces a row-wise or horizontal subset of the relation. For example, a selection requiring the specimen number to be less than 500 from table 2 would give the result found in table 3.

labvals			
spec#	Hgb	MCV	RBC_morph
423	16.5	85	anisocytosis
425	11.1	78	microcytosis
455	10.4	77	microcytosis

Table 3. Results of Selection Operation

Projection

Projection, on the other hand, provides a column-wise or vertical subset of the relation. Redundant tuples are eliminated from the resultant relation. For example a projection of the RBC_morph column would yield table 4.

labvals
RBC_morph
microcytosis
normal
poikilocytosis
anisocytosis

Table 4. Results of Projection Operation

Join

Join is slightly more complicated than selection and projection. It produces a combination of two (or more) tables based on all attributes they have in common. There are really several different kinds of join, the one referred to here being the *natural join*. The resultant table will have a tuple for each pair of tuples in the original tables that share identical attribute-values for every attribute the tables share. If the original tables have no attributes in common then the resultant table is the cartesian product of the two tables. If no pairs of tuples have identical attribute-values (assuming a common attribute) then the join results in a null table. For examples, see section 4.2.1. and tables 8 and 9.

The operations above can be incorporated into a powerful retrieval language called a relational calculus. The following chapter describes the fundamental constructs of the language QUIN uses as such a calculus and retrieval language.

CHAPTER 4

Data Language VL

This chapter describes the capabilities and use of the VL data language used by QUIN. VL instructions provide the capabilities for relational table creation, retrieval and modification. The language is easily learned and requires a minimum of procedural specification so that it is reasonable to expect that users with minimal computer background could quickly learn and use it.

4.1. Table Creation

The instructions for creation of tables are **define** and **add**. **Define** creates an empty table and sets up the specifications for the attribute list and the key, while **add** places new tuples into a table.

4.1.1. Define

This instruction specifies a new table, its name, the names of the attributes, and (optionally) the name(s) of the key(s). The names of tables and attributes must begin with a letter and can contain any combination of letters, numerals, and the characters "#" and "_". No two tables may have the same name, nor can a table-name be the same as any attribute-name or reserved word. A table may not have two identical attribute-names, but two different tables may (and often do) share a common attribute-name. Keys are optional but if declared they should be the first (i.e. leftmost) attribute(s) in the table.

Consider as an example the definition of a database that keeps track of results of blood tests on patients, as in the example in the previous chapter, table 2. The table, called "labvals", stores the information on specimens and the values measured. The unique attribute (key) of each record would be the specimen number. The way to create this table using the **define** instruction would be:

```
define labvals (spec#, Hgb, MCV, RBC_morph)  key := spec#
```

We could also define a table called "spec" to keep track of the dates of individual blood specimens:

```
define spec (spec#, ID#, day, month, year)  key := spec#
```

Another table called "ptrc" would store a patient's identification number, his name, and his admitting diagnosis:

```
define ptrc (ID#, name, dx)  key := ID#
```

4.1.2. Define Event

An event is a table with only one row. Its purpose is to specify a complete single data object with attributes that may be found in several different tables, so that adding the event to each of those tables is easier and more reliable. Events are defined by the **define event** instruction followed by the event name and then a parenthesized list of attribute-value pairs. Continuing the previous example, we could define an event that contains all the attributes of the three tables (ptrc, spec and labvals):

```
define event E1
    (ID# := 988,
     name := Jones,
     dx := iron_deficiency,
     spec# := 1024,
     day := 25,
     month := 6,
     year := 1982,
     Hgb := 10.3,
     MCV := 78,
     RBC_morph := microcytosis)
```

Event "E1" records that a blood test was done on patient number 988 whose name is Jones and whose diagnosis was iron_deficiency. The blood, specimen number 1024, was drawn on 25 June 1982 and the results showed a hemoglobin of 10.3, a mean corpuscular volume of 78, and red cell morphology was microcytosis. The attribute-value pairs in the event definition can be arranged in any order.

4.1.3. Add

The add instruction places tuples (rows) into a table. There are four forms of the instruction: one for single row addition, another for multiple rows, one for adding an event to a table, and one for adding an external file of tuples to a table.

(1) A single row may be added as follows:

```
add (365, Smith, aplastic_anemia)  to ptrc
```

(2) Multiple rows are added in similar fashion:

```
add to ptrc
(398, Clark, folate_deficiency)  (404, Blake, iron_deficiency)
(425, Smith, hemolytic_anemia )  (241, Jones, iron_deficiency)
end
```

(3) Adding an event to several tables is simple:

```
add E1 to ptrc
add E1 to spec
add E1 to labvals
```

(4) Adding an external file named "vls" to the "labvals" table would be done as follows:

```
add vls to labvals
```

The external file must be set up in tabular form. For example, the file "vls" might appear as in figure 4.

```
891   13.1   90   Normal
555   14.2   88   Poikilocytosis
423   15.5   85   Anisocytosis
```

Figure 4. Format of File "vls"

Addition of tuples may be done at the beginning of the table, the end, or before or after any specified row in the table by using a *row condition.* All four forms of add may be used with a row condition. If no row condition is specified then the addition is done after the last row of the table. For example, the following places a new tuple at the first row:

```
add (425,404,26,6,1982) to specs : [row<1]
```

The colon is to be read "such that" and the row condition is of the same form as retrieval conditions (see section 4.2.2). The reserved word *last* may be used to insert before the last row:

```
add datafile to specs : [row<last]
```

The condition "[row>last]" would be redundant because that is the default. If there are not as many rows in the table as specified in the row condition, the new tuples will be added at the end of the table.

4.2. Table Retrieval

The retrieval commands are **get** and **let**. Simple retrieval of an entire table requires only listing the table name after the keyword **get**. Selected portions of the table can be retrieved and displayed also (see sections 4.2.1. and 4.2.2.). A new table can be created with the keyword **let** followed by the name of the new table, ":=", and the description of the new table. For example, the following command creates a new table called "tests" which is the same as the "labvals" table:

```
let tests := labvals
```

The new table would be created but not displayed. The command to display the table is:

```
get tests
```

Table 5 would then be displayed. It would be identical to the "labvals" table except for its name.

tests			
spec#	Hgb	MCV	RBC_morph
1024	10.3	78	microcytosis
891	13.1	90	normal
555	14.2	88	poikilocytosis
423	16.5	85	anisocytosis
425	11.1	78	microcytosis
455	10.4	77	microcytosis

Table 5. Result of Retrieval

The combined effect of the get and let could have been accomplished by simply saying:

```
get tests := labvals
```

The let command is used to create temporary tables that can be used as working copies or can be saved as new permanent tables. It never displays the results of its work. The get command always displays its results, and can also be used to create temporary tables. A wide variety of more complicated retrieval instructions can be specified by appending the appropriate modifying expressions to these two commands. These additional expressions are specified by a *relational table expression* (see section 4.2.1.) optionally followed by a *VL condition* (see section 4.2.2.).

4.2.1. Relational Table Expressions

The table expression specifies the table or tables and the attribute or attributes to be retrieved. All operations described in section 3.4 except selection can be specified with a table expression. When more than one table is listed in an expression, tables must be separated by an operator. The operator symbols and their meanings are given by table 6.

Symbol	Meaning
*	Join
v	Union
&	Intersection
-	Difference
+	Append

Table 6. Relational Table Expression Operators

These operators take precedence over all others in the retrieval instruction. All tables listed together with the logical operators (union, intersection, difference) and the append operator must be *union compatible*, which means that they must have identical lists of attributes, in the same order, and of the same type (e.g. if attribute N is an integer in one table, it must also be an integer in the other table(s) in the instruction).

To illustrate the use of operators in table expressions, let us assume we have a table named "spec" as shown in table 7.

spec				
spec#	ID#	day	month	year
1024	988	25	6	1982
425	404	26	6	1982
850	405	27	6	1982
455	406	27	6	1982

Table 7. The "spec" Table

The following instruction would create a table called "T1" which is a join of the "spec" and "labvals" tables (the common descriptor being the specimen number):

```
let T1 := spec * labvals
```

The join of tables "spec" and "labvals" would appear as in table 8.

T1							
spec#	ID#	day	month	year	Hgb	MCV	RBC_morph
1024	988	25	6	1982	10.3	78	microcytosis
425	404	26	6	1982	11.1	78	microcytosis
455	406	27	6	1982	10.4	77	microcytosis

Table 8. Join of "spec" and "labvals"

If no attributes are listed in the retrieval command (as in the example above) then the full set of attributes for all tables is retrieved. If a subset of attributes is specified then the **projection** of those attributes on the table is retrieved. For example:

get spec * labvals (ID#, Hgb)

will retrieve a table with two columns. It will be the projection of ID# and Hgb on the join of spec and labvals. Table 9 shows the result.

ID#	Hgb
988	10.3
404	11.1
406	10.4

Table 9. Result of Projection and Join

For the purposes of the inference algorithms it is sometimes not desirable to eliminate redundancy when doing projection, so QUIN provides two other methods of specifying projection. One method, using &, simply does a "column selection" and retrieves all rows even if redundant. The other method, using #, eliminates redundancy but provides an additional column that shows the number of times that a particular row occurs. All three forms of projection instruction are illustrated in figures 5, 6, and 7.

get labvals(RBC_morph)

labvals
RBC_morph
microcytosis
normal
poikilocytosis
anisocytosis

Figure 5. Ordinary Projection

---

get labvals(RBC_morph,#)

labvals	
RBC_morph	#
microcytosis	3
normal	1
poikilocytosis	1
anisocytosis	1

Figure 6. Projection with Count

---

get labvals(RBC_morph,&)

labvals
RBC_morph
microcytosis
normal
poikilocytosis
anisocytosis
microcytosis
microcytosis

Figure 7. Projection with Redundancy

An attribute may be replaced by a function of an attribute in the retrieval expression. Available functions include **min**, **max**, **sum**, **count** and **domain**. Figure 8 gives an example of the use of the **min** function.

```
get labvals( min(Hgb) )
```

labvals
Hgb
10.3

Figure 8. Use of "min"

4.2.2. VL Conditions

A VL *condition* is the part of a retrieval command which specifies **selection**. The following example illustrates the major features of a VL condition. The condition begins with a colon which should be read "such that."

```
get labvals : [Hgb = 14..16] [RBC_morph <> normal] v [Hgb < 14]
```

The command would retrieve all rows from table "labvals" in which either a) the Hgb is in the range 14 to 16 and the RBC_morph is not normal, or b) the Hgb is less than 14.

A VL condition thus consists of a disjunction of one or more *complexes*. Complexes consist of a conjunction of one or more *selectors*. In the instruction above, "[Hgb = 14 .. 16]" is a selector. Selectors may be separated by the conjunction operator &, or simply listed one after the other, as in the complex "[Hgb = 14..16] [RBC_morph <> normal]" above. Selectors or groups of selectors (complexes) may be separated by the disjunction operator v. Thus a condition is a "sum of products" of logical (VL) selectors.

VL selectors are used to specify the body of the condition. They can be of two types, row-oriented selectors ("tuple calculus") and set-oriented selectors ("domain calculus"). The example above uses row-oriented selectors.

A row-oriented selector consists of a left square bracket, an attribute name (the referee), a comparison operator (=, <>, <, >, <=, >=), a comparison value (the reference), and a right square bracket. The comparison value may be a single value (e.g. "normal"), an arithmetic range of values (e.g. "14 .. 16"), an arithmetic expression (e.g. "Hgb + 2.5"), or a list of values, ranges, or expressions separated by the "or" operator (e.g. "3..5 v 7 v Hgb/10").

A domain-calculus selector consists of a referee set, a set comparison operator and a reference set. The referee set is called an *image set* of the attributes being retrieved. An image set is the set of values of the image attribute (or of unique combinations of values of the image attributes) corresponding to each retrieval value (or unique combination of retrieval values). The comparison operator is the same as those in the tuple-calculus selector (using =, <>, <, >, <=, >=) but the meanings are set comparisons instead. Thus "=" tests set equality and "<" tests to see if the first set is a proper subset of the second. The reference set has the same attribute list as the referee set but may contain a VL condition within it, as in this example:

```
get spec(ID#) : {day,month,year} = {day,month,year:[ID# = 365]}
```

Day, month, and year are the image attributes. {day,month,year} specifies the referee set. It may also be written {day,month,year : ID#}, which reads "the set of day-month-year triples corresponding to each ID#." It is calculated anew for each unique value of ID#, the retrieval attribute. [ID#=365] is the VL condition within the reference set. The meaning is that the user wants to retrieve the ID# of all patients who had specimens done on precisely the same days as patient number 365. The same thing could be accomplished using the tuple-oriented calculus only by repeatedly doing the following for every value of n:

```
get spec(day,month,year):[ID# = n]
```

The results would then have to be compared with the result of:

```
get spec(day,month,year):[ID#=365]
```

and those values of ID# for which the results matched that of 365 would be the final result.

4.2.3. Ordering of Rows

All retrievals may optionally have an ordering condition. The phrases "order up on" and "order down on" are appended to the retrieval instruction, along with the name of the attribute to be ordered on. For example, the instruction

```
get ptrc order up on ID#
```

will retrieve the table "ptrc" in ascending order of ID numbers.

4.3. Table Modification

The table modification instructions are **change**, **delete**, and **save**.

4.3.1. Change

The **change** instruction is used to assign new values to existing rows in a table or to change the name or type of an attribute. When the user types:

```
change tablename
```

he enters a "sub-instruction" mode in which all commands refer to the table being changed. A working copy of the original table is made for security in case of error, and the user's prompt is ">>". To leave the change mode the user types **abort** or **exit**, with only the latter exit resulting in actual modification of the original table.

There are several commands available in the change mode. The user may use ordinary assignment statements to change the values of each attribute. Some examples are given below. The assignments may be followed by a VL condition that restricts the assignment of values to specific rows of the table. Attribute names may be changed by specifying the condition "[row=0]". The **display** sub-instruction displays the working table; the **get** sub-instruction displays the original table before any changes. A simple **change** instruction sequence may be entered as shown below with table 10:

```
> change labvals   /* enter change sub-mode */
ok                 /* system response */
>> get             /* look at original table */
```

labvals			
spec#	Hgb	MCV	RBC_morph
1024	10.3	78	microcytosis
891	13.1	90	normal
555	14.2	88	poikilocytosis
423	16.5	85	anisocytosis
425	11.1	78	microcytosis
455	10.4	77	microcytosis

Table 10. Original Table to be Changed

The commands below with table 11 illustrate how the table might be modified within the change mode.

```
>> Hgb := high :[Hgb>16]
>> Hgb := low  :[Hgb<14]
>> Hgb := normal :[Hgb=14:16]
>> display   /* look at changed table */
```

change$table			
spec#	Hgb	MCV	RBC_morph
1024	low	78	microcytosis
891	normal	90	normal
555	normal	88	poikilocytosis
423	high	85	anisocytosis
425	low	78	microcytosis
455	low	77	microcytosis

```
>> end              /* exit, making the changes permanent */
change completed   /* system response */
```

Table 11. Table with Modified Hgb Column

4.3.2. Delete

Delete is used to remove rows or columns from a table, or to remove a table from the database. Each of these three functions is accomplished by a different form of the instruction.

(1) Deleting rows is accomplished by specifying a VL condition:

```
delete ptrc : [ID# = 250 .. 500]
```

This will delete all rows in table "ptrc" where the ID number is in the range 250 to 500 inclusive.

(2) Deleting columns is accomplished by specifying a projection:

```
delete spec(day,month)
```

This will delete the day and month columns from table "spec".

(3) Deleting an entire table or event is done by simply giving the table name:

```
delete T1
```

This will remove the table named "T1" from the database.

4.3.3. Save

Any tables created with the **get** or **let** instructions will be given temporary status; **save** is the instruction that changes temporary to permanent status. Tables with permanent status will stay in the database after a session is completed, whereas tables with temporary status will be deleted at the end of a session.

4.4. Help

The system has on-line help available to describe the use of each command. Help can be obtained by typing help or by simply typing a question mark at a prompt. If specific information is desired about a particular command, the command name should be entered following the word "help," followed by a carriage return.

CHAPTER 5

Inferential Operations

This chapter describes the inferential operations on an informal conceptual level. References are provided for more detailed explanations of the algorithms and the theories supporting them. In the current implementation, only cluster and diff are operational. However, the same methods of interaction can, in principle, be used with all the inference programs mentioned here.

5.1. Cluster

The purpose of the cluster operation is to divide a collection of objects into smaller groups of similar objects based upon some criterion or measure of similarity. Clustering is the process of developing a taxonomy or classification scheme for the objects of a study.

The program invoked by the cluster command in QUIN is called *CLUSTER/paf* [14]. The reader is urged to consult the references cited for more complete explanations of the details of the program's operation and theoretical background. Unlike most numerical taxonomic techniques, this program uses a "concept-based" method of clustering that produces descriptions of the clusters (categories) that it derives. It also permits the user to specify the criteria which are to be used to evaluate clusters. One or several criteria can be maximized simultaneously to produce the optimal clusters. Some of the criteria available to characterize clusters include:

- the fit between the clustering and the data (sparseness),
- the total inter-cluster differences (degree of intersection),
- the number of attributes which singly distinguish between all the clusters (essential dimensionality), and

- the simplicity of cluster descriptions (number of selectors).

The names and numbers of criteria currently available appear in table 12.

criterion number	brief description
1	sparseness
2	degree of intersection
3	number of events occurring in more than one complex
4	share of events (evenness of cluster size)
5	number of selectors (simplicity of cluster descriptions)
6	essential dimensionality (dimensionality of differences)
7	relevant-variable sparseness
8	relevant-variable-set sparseness

Table 12. Clustering Criteria

The cluster operation is invoked by the following instruction,

```
cluster (events,parameters,results)
```

where "events" and "parameters" represent the names of relational tables within QUIN, and "results" is the name of a table which may or may not already exist (if not it will be created). Any legal table names may be used in the command. The events table must contain the descriptions of the objects to be clustered with each object occupying one row in the table. Each column represents an attribute of the objects in the table. The parameters table is used to indicate *K* (the number of clusters to be formed), the criteria to be used and other optional parameters. The optional results table is the table in the database to which the results of the clustering will be returned. If no results table is specified, the output of cluster can be found in a file in the user's working directory.

A simple example of clustering follows, using data similar to the previous examples in chapter four. In addition to the "events" table, a "parameters" table and at least one "criterion" table must be prepared before issuing the cluster command. Table 13 illustrates a parameters table and table 14 illustrates a criterion table.

parameters	
k	criterion
3	crl
2	crl

Table 13. Parameters Table

crl_criterion	
criterion	tolerance
1	0.0

Table 14. Criterion Table

The parameters table (table 13) has two tuples (rows). The cluster algorithm will therefore be run twice, once for each row in the parameters table. The first time it will split the events into three groups (k=3) and the second time it will create only two groups (k=2). Criterion "crl" (found in table 13) is defined in the criterion table called "crl_criterion" (table 14). Criterion "1" is *sparseness*, (see table 12). It is the only criterion which will be used by the program in this example. The tolerance is a measure of the degree of error allowed in fitting the clusters to the criterion.

The events table for this example is shown in table 15.

events		
mcv	hgb	mchc
0	1	0
2	2	0
1	1	1
3	4	3
3	4.	3
4	1	4
5	2	4
4	1	4

Table 15. E[illegible]s to be Clustered

The values in the events table must all be integers. Attributes, such as those represented here, which ordinarily have continuous linear values must be made discrete before cluster can deal with them. The meanings of the values in the events table are given in figure 9, with the values from the events table in the "#" column followed by the range of real values which have been assigned to each value.

#	mcv	
0	<60	
1	60 to 69	
2	70 to 79	
3	80 to 94	(mcv normal = 80 to 94 cu microns)
4	95 to 104	
5	>104	

#	hgb	
0	<8 = 0	
1	9 to 10	
2	11 to 12	
3	13 to 14	
4	15 to 16	(hgb normal = 14 to 18)
5	17 to 18	
6	19 to 20	
7	>20	

#	mchc	
0	<27	
1	27 to 32	
2	33 to 38	(mchc normal = 33 to 38 %)
3	39 to 44	
4	>43	

Figure 9. Meanings of Values in Events Table

When this sample is run, cluster splits the events into three groups as in figure 10.

```
Group one   : events 1,2,3
Group two   : events 4,5
Group three : events 6,7,8

Group one is described as:
  mcv<80 and hgb=9..12 and mchc<33
Group two is:
  mcv=80..94 and hgb=15..16 and mchc=33..38
Group three is:
  mcv>94 and hgb=9..12 and mchc=39..44.
```

Figure 10. Results of Clustering

This is a particularly simple example, but it gives the flavor of the clustering operation. In this case, cluster has discovered three groups which can be interpreted as being cases of microcytic anemia (group one), normal blood counts (group two), and macrocytic anemia (group three). For further examples of the use of cluster see [22].

5.2. Diff

Diff (differentiate) takes a number of classes of events that have already been categorized, and attempts to find the conceptually simplest rules that will predict the category of each event, i.e. discriminate between the categories. The algorithm invoked by the command is called *Aq* and is incorporated in the program called *GEM* [15].

The following is an example of the use of the diff instruction to create rules for differentiating the groups of objects represented in three tables named grp1, grp2 and grp3. There are a number of differences from the cluster operation:

(1) No criterion tables are needed.

(2) The parameters table, an example of which is shown in table 16, has a different format from the cluster parameters table which was illustrated in table 13.

params	
echo	maxstar
pcve	10

Table 16. GEM Parameters Table

(3) The values in the events tables need not be integer only. Discrete nominal values are allowed in addition to integers. QUIN automatically generates the other input tables required by GEM. An example of the way an events table might appear is given in table 17.

grp1		
day	rainfall	hours_sunlight
Monday	light	6
Saturday	none	12
Wednesday	heavy	2

Table 17. GEM Events Table

Note that there still must not be real-valued attributes (e.g. 50.2) but nominal attributes are allowed. In addition, the type and domain of an attribute can be declared for the use of "diff." In the table above, the "day" attribute clearly can take on seven values which are ordered and cyclic. Only three of these appear in the table and they are not in order. To define the domain of this attribute one must first prepare a relational table with one column containing all possible values which the attribute may have. The values should be listed in order, as in table 18.

weekdays
names
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday

Table 18. Domain Values for Days of the Week

Given such a table, the following command will set up a permanent domain for the attribute "day" which will automatically be referred to whenever the system needs to prepare "day" as an attribute for the "diff" operation:

```
domain (day) := weekdays
```

The "diff" operation recognizes three types of attributes :

- **nominal** (discrete unordered)
- linear (discrete ordered, such as "rainfall" in table 17), and
- cyclic (discrete with cyclical ordering, such as days of the week).

The system assumes that integer values are linear and that alphabetic values are nominal. When the opposite is true or when the attribute is cyclic, the following instruction can be used to define the type of the attribute for use by "diff:"

```
type (day) := cyc
```

The abbreviations for nominal, linear and cyclic are **nom**, **lin** and **cyc**, respectively.

The instruction format for invoking "diff" is as follows:

```
diff(grp1, grp2, grp3; params, rslts)
```

The "params" and "rslts" tables are optional. The system provides default parameters to GEM if a parameters table is omitted. If the "rslts" table is included, the discrimination rules produced by GEM will be placed in it. A variable number of groups (event tables) may be submitted. A semicolon indicates the end of the list of event tables, as after grp3 above.

5.3. Esel

The operation **esel** * invokes Esel [17], a program that takes a large number of examples and selects a small subset of examples that is most representative of the larger group. The smaller sample will require less execution time in inference programs such as CLUSTER/paf or GEM. Very large numbers of examples (more than 200) would probably require inordinate amounts of processing time, making it useful and efficient to choose a representative subset.

5.4. Varsel

The **varsel** * instruction invokes a program called PROMISE [16] which selects the most "promising" attributes for differentiating between classes of events. Its output is therefore intended for use with GEM. The elimination of irrelevant attributes is a horizontal reduction of the database somewhat comparable to the vertical reduction accomplished by esel. The reduction results in reduced execution time in GEM but also results in the elimination of attributes from consideration by the inference process.

5.5. Varcon and Varcont

The **varcon** * instruction (variable construction) invokes a program called NEWVAR [18] which attempts to use mathematical operations (multiplication, addition) to create new attributes from combinations of existing attributes. The use of ratios or differences of existing attributes sometimes provides simpler and more accurate rules for distinguishing one class from another.

The command **varcont** * is used to access a program named CONVART [19], a system for inducing time-dependent information from data. Multiple measurements of an attribute over time can be changed into a single attribute based upon its time-dependent characteristics. The induced description of the time-dependent attribute can then be used in data for input to other inference routines.

5.6. Other Operations

The **apply** * operation tests the performance of induced rules on new events. It currently a part of the AQ11 [20] program. The output is a confusion matrix that gives the percentage of false positive and false negative decisions for each decision category.

Another inference operation, **treecon** * uses program OPTREE [21] to produce optimal decision trees from extended entry decision tables [23]. It performs the conversion of VL rules to decision trees (branching logic) for the convenience of the user.

There are three low-level inference operations that are used in CLUSTER/paf that also could be invoked separately.

(1) **Sim** * (similarity) takes any two events and calculates a syntactic similarity measure. The similarity of two events is the inverse of the syntactic distance measure used in CLUSTER/paf.

(2) **Refun** * (reference union) takes the values of attributes and "collapses" several events into one event with multiple-valued attributes. For example, the events

(12, medium)
(13, large)

could undergo reference union to become

(12 v 13, medium v large).

(3) **Gen** * (generalize) goes one step further to take the events that result from reference union and generalizes the results into more intuitively succinct values. Thus

(12 v 13 v 14 v 15, medium v large v verylarge)

would become

(12..15, >small).

There are other inference programs that may be useful tools for the generation of knowledge from examples, and they too have the potential to be integrated into the system.

* planned operation not currently implemented

CHAPTER 6

Conclusion

This chapter provides a brief subjective evaluation of QUIN as a tool for management and analysis of data, including some of its strengths, weaknesses, and ideas for further work.

6.1. Strengths

The strengths of a database analysis system have been referred to in previous chapters. The automated storage and retrieval of data facilitate the researcher's ability to prepare input to inference programs; the inference programs provide tools unavailable in standard statistical packages for the conceptual analysis of data. The integration of several inference operators with the database appears to offer opportunities for conceptual data analysis not previously feasible.

The VL data language is easily learned and provides a less wordy query language than most relational algebras. It compares favorably with many relational calculus languages, and appears to provide a reasonable means of querying without the need for a complicated natural language interface. The language is intended to be easy enough to make the inference operations available to a wider range of people who do not have extensive experience in computer science.

6.2. Weaknesses

QUIN has several weaknesses as a database management system. Some of these weaknesses stem from the adoption of the relational model of data. For example, medical records are often very sparse and may not contain recorded values for large numbers of disease attributes. The recording of some values often depends on the presence of others, resulting in a structure that is more naturally hierarchical. Other weaknesses stem only from the limited scope of the current implementation that could easily be extended and revised:

- QUIN is a program, not a programming product. The goal of its construction to this point has been to create a research tool that works; the eventual goal is to have a system that is free of bugs and convenient to use.
- Data entry has not been optimized for naive user input. Any typing mistakes made during command entry will result in rejection of the command, and there is no facility for structured or prompted relational table entry or edit.
- The time required for retrieval commands to complete can exceed the acceptable time for interactive use of the system for large tables (greater than 100 rows). QUIN was not written with efficiency as a major design criterion; this means that there are likely several places in the code in which optimization of the procedures may produce more rapid execution.
- Many database management systems have mechanisms for handling backup, security, variable user access privileges, report generation and other useful facilities which were beyond the scope of this work.

6.3. Ideas for Future Work

The wide variety of inference programs included as planned operations in QUIN suggests that there is much yet to be learned about strategies for using these operations and about the ways in which they interact. It is now possible to begin to explore the application of QUIN to practical problems.

There are a few descriptive statistical functions in the query language and these could be extended to include more statistical tests. QUIN could also be integrated with standard statistical packages that take data input in tabular form. This could be done in a manner similar to the inference program interface.

As a database QUIN would become more friendly, usable and sophisticated if the user could specify semantic constraints on the data and if the system had a convenient method for interactive data entry, such as a frame-based screen editor.

In summary, QUIN is regarded as a significant first step in providing a convenient interface to many inductive inference techniques that operate on large numbers of examples. The use of this tool is expected to provide useful insights into the process of machine inference and the evaluation of inferred knowledge.

APPENDIX A

Program Description

QUIN is written in Berkeley PASCAL for the UNIX operating system. The 'qq' prefix is attached to every procedure in QUIN to avoid name conflicts with other ADVISE modules. There are three major files of PASCAL code (named qqmain.p, qqparsr.p and qqex.p) and a file of C code for operating system calls (qqcfunc.c). Qqmain.p contains the initialization routines, the session handling routines, and utility routines. Qqparsr.p contains the command parser, and qqex.p contains the command executor. There are also other small pieces of code (qqprocs.h and qqconst.h), containing the external declarations of procedures which are accessed by more than one segment, and the constant, type, and variable declarations as well as the definition of the qqstatic area (see description of ADVISE [1] for more on static areas).

Flow of control through the code follows the pattern of

1) start in the session routine and initialize the database,

2) parse a command into PT (the parse table - see appendix C),

3) go to the executor and execute the command,

4) cycle back to number 2.

When inference commands are given, there are two options for the invocation of the inference program. If the session is batch mode or if the user specifies a "results" table, then the inference program is forked and QUIN waits for it to finish before continuing. In other circumstances, (interactive and no results table specified) QUIN does not wait for the inference program to complete. In any circumstance, the output of the inference programs can be found in a file in the user's working directory.

Network Data Structures

The relational table data structures are handled by the ADVISE memory management routines. The following details of the network form of tables are in the ADVISE network text format [1]. A network stores a tuple with the internal name of every table in the network under a node called QQDEFAULT with an arcname called QQTABLES. All attributes are also stored in a tuple under the same node with an arcname called QQDES (for "descriptors," another name for attributes). In the future, other information about the relational tables in a network could also be stored under the QQDEFAULT node. An illustration of the format of the node follows:

```
(QQDEFAULT (
  (QQDES P# pname color weight S# sname status city qty )
  (QQTABLES Parts Suppliers SP ) ) )
```

Tables in the network are stored under the node corresponding to the name of the table. The following is an example of a table named "Parts" as it would appear in the network:

```
(Parts (
  (QQWIDTH 2 6 5 6 )
  (QQTYPE 3 3 3 1 )
  (QQKEY 1 )
  (QQDES P# pname color weight )
  (*EOF* )
  (MKVALTUPLE P1 nut red 12 )
  (MKVALTUPLE P2 bolt green 17 )
  (MKVALTUPLE P3 screen blue 17 )
  (MKVALTUPLE P4 screw red 14 )
  (MKVALTUPLE P5 cam blue 12 ) ) )
```

The first four slots under the node contain information about column print widths, attribute types, keys and descriptors. The arcnames for these tuples are QQWIDTH, QQTYPE, QQKEY and QQDES respectively. These four tuples may be in any order. They are separated from the "real" tuples of the table by a tuple with arcname "*EOF*." This separation is important because QUIN calculates row numbers using the slot number of *EOF* as a reference. If the table main node is to have other tuples attached to it, they should be added before the *EOF* ; QUIN can then dynamically adjust its calculation of the correspondence between slot number and row number. Tuples containing values have arcname MKVALTUPLE.

APPENDIX B

Grammar for Data Language VL

This is the grammar of the QUIN sublanguage grammar as it currently is defined. The VL grammar was originally written by Richard Schubert and revised by Albert Cheng. I have made further revisions as well, but the nature of the grammar has remained essentially the same. The parser for the language is a hard-coded deterministic parser. The actual implementation is therefore only a subset (nearly complete) of the language specified in this appendix. The following conventions are used for the specification of the grammar:

{ } means optional

{ }* means zero or more

{ }+ means one or more

" " surrounds special characters

| separates alternatives

```
SESSION ::= {COMMAND}* "exit"
COMMAND ::= CREATE | RETRIEVE | MODIFY | SAVE | COMMENT | HELP | INFER
CREATE ::= DEFINE | ADD
DEFINE ::= "define" DEF-LIST
DEF-LIST ::= {"rt"} RTNAME "(" ATTR-LIST ")" {"key :=" ATTR-LIST}
        | "event" EVENT
EVENT ::= EVENTNAME ":=" "(" ATTR ":=" VALUE {"," ATTR ":=" VALUE}* ")"
ADD ::= "add to" RTNAME {":" ROW-COND} {VALUE-LIST}+ "end"
        | "add" EVENTNAME "to" RTNAME {":" ROW-COND}
        | "add" VALUE-LIST "to" RTNAME {":" ROW-COND}
        | "add" FILE-NAME "to" RTNAME {":" ROW-COND}
VALUE-LIST ::= "(" VALUE {"," VALUE}* ")"
ROW-COND ::= "[row>" ROW-VAL "]"
        | "[row<" ROW-VAL "]"
ROW-VAL ::= VALUE | "last"
RETRIEVE ::= "get" {LABEL ":="} RT-COND {ORDER}
        | "let" LABEL ":=" RT-COND {ORDER}
LABEL ::= RTNAME
RT-COND ::= RT-EXPR {CSYM CONDITION}
        | ATTR-LIST {CSYM CONDITION}
CSYM ::= ":" | "where"
RT-EXPR ::= TABLE-EXPR "(" ATTR-LIST ")" | JOIN-EXPR | ATTR-LIST
TABLE-EXPR ::= RTNAME {TABLE-OP RTNAME}*
TABLE-OP ::= "+" | "-" | "*" | "&" | " v "
JOIN-EXPR ::= "(" DOT-EXPR {"," DOT-EXPR}* ")"
        | RT-PROJECT {"," RT-PROJECT}*
DOT-EXPR ::= RTNAME "." ATTR
RT-PROJECT ::= RTNAME {"(" ATTR-LIST ")"}
CONDITION ::= VL-COMPLEX {" v " VL-COMPLEX}*
VL-COMPLEX ::= SELECTOR {{"&"} SELECTOR}*
SELECTOR ::= TUPLE-SELECTOR | DOMAIN-SELECTOR
TUPLE-SELECTOR ::= "[" ATTR RELOP VALUES {" v " VALUES}* "]"
DOMAIN-SELECTOR ::= "{" DOMAIN-VAR "}" RELOP "{" RT-COND "}"
DOMAIN-VAR ::= ATTR-LIST CSYM ATTR-LIST
VALUES ::= VALUE {".." VALUE}
VALUE ::= AEXPR | NAME | UNKNOWN
```

```
UNKNOWN ::= "?" | "*" | "$"

AEXPR ::= {AEXPR "+"} ATERM | AEXPR "-" ATERM

ATERM ::= {ATERM "*"} AFACTOR | ATERM "/" AFACTOR

AFACTOR ::= CONSTANT | "(" AEXPR ")" | "-" AFACTOR

ORDER ::= "order up on" ATTR | "order down on" ATTR

ATTR-LIST ::= ATTR {"," ATTR}*

ATTR ::= ATTRIBUTE-NAME | DOT-EXPR | FUNCNAME "(" ATTR ")"

FUNCNAME ::= "min" | "max" | "sum" | "count" | "domain" | "ave"

MODIFY ::= "change" RTNAME "cr" {CHANGE-STMT}* END-CHANGE

CHANGE-STMT ::= ASSIGNMENT | NEW-ATTR | "get" | "display"

END-CHANGE ::= "end" | "abort"

ASSIGNMENT ::= ATTR ":=" VALUE {CSYM CONDITION}

NEW-ATTR ::= ATTRIBUTE-NAME ":=" LABEL ":[row=0]"

DELETE ::= "delete" RTNAME | "delete" RTNAME "(" ATTR-LIST ")"

        | "delete" RTNAME CSYM CONDITION

SAVE ::= "save" RTNAME {"," RTNAME}*

COMMENT ::= "comment" {ANYTHING} "end"

HELP ::= "?" | "help" | "help" INSTRUCTION

INSTRUCTION ::= "define" | "add" | "get" | "let" | "change"

        | "delete" | "exit" | "comment" | "help" | INF-OP

INFER ::= INF-OP "(" EVENTS {"," EVENTS}* "," PARAMETERS {"," RESULTS} ")"

EVENTS ::= RTNAME

PARAMETERS ::= RTNAME

RESULTS ::= LABEL

INF-OP ::= "cluster" | "diff" | "esel" | "varsel" | "varcon"
```

APPENDIX C

Parse Table Layout

The Parse Table (PT) is an array of integers which is used for communication between the VL data language parser and the command executor. The integers may represent pointers to an array of character strings (symtable), to an array of real numbers (realn), to a tuple of relational table internal names (qqsptr^.tbl) or to a tuple containing all attributes in the database (qqsptr^.des). The following describes the semantics of the PT array positions for each of the commands and conditions in the language.

COMMAND	ARRAY POSITION	MEANING
DEFINE		
	1	Length of PT
	2	1 (DEFINE)
	3	Length of PT used
	4	Number of tables or events defined
for each table defined:		
	x	Length of PT used for this table
	x + 1	1 (DEFINE TABLE)
	x + 2	Index in symtable of table name
	x + 3	Number of attributes (na)
	x + 4	Number of keys (nk)
	x + 5	1st attribute
	x + 6	2nd attribute
	.	
	.	
	.	
	x + 4 + na	Last attribute
	x + 4 + na + 1	1st key
	x + 4 + na + 2	2nd key
	.	
	.	
	.	
	x + 4 + na+ nk	Last key
for each event defined:		
	x	Length of PT used for this event
	x + 1	2 (DEFINE EVENT)

x + 2	Index in symtable of event name
x + 3	Number of attributes (na)
x + 4	1st attribute
x + 5	1st value
x + 6	1st type
x + 7	2nd attribute
x + 8	2nd value
x + 9	2nd type
.	
.	
.	
x + 3(na) + 1	Last attribute
x + 3(na) + 2	Last value
x + 3(na) + 3	Last type

ADD

1	Length of PT
2	2 (ADD)
3	Length of PT used
4	Type of add : 1: add to 2: add event 3: add (value, value, ...) to 4: add file to
5	Table number to be added to
6	Row condition : 0: none 1: < (before) 2: > (after)
7	Row number (-1 if none given)

for add types 1 and 3:

8	Number of rows to be added (ne)
9	Number of columns in the rows (na)
10	1st column's type
11	2nd column's type
.	
.	
.	
9 + na	Last column's type
9 + na + 1	1st value
9 + na + 2	1st value's type
9 + na + 3	2nd value
9 + na + 4	2nd value's type
.	
.	
.	
9+ na+ 2(na)(ne)	Last value's type
" + 1	1st column's maximum print width
" + 2	2nd column's maximum print width
.	
.	

.	
9+ 2(na)(ne+ 1)	Last column's maximum print width

for add type 2 (event add):

8	Event to be added

for add type 4 (file add):

8	Index in symtable of file's name

CHANGE

1	Length of PT
2	3 (CHANGE)
3	Length of PT used
4	Table number of table to be changed

while within the change statement:

5	Change commands: 1 : Assignment 2 : Display (change table) 3 : Get (original table) 4 : End 5 : Abort 6 : New attribute name

for assignment:

6	Descriptor number
7	Length of PT used for value expression
8	Value expression
.	.
.	.
.	.
8 + pt[7]	Length of PT used for selection expression
8 + pt[7] + 1	Selection expression
.	.
.	.
.	.

for new attribute name:

6	Descriptor number
7	3 (length of this expression)
8	2 (attribute)
9	0 (no meaning)
10	New attribute number

DELETE

1	Length of PT
2	4 (DELETE)
3	Length of PT used

Table expression (see below)

HELP

	1	Length of PT
	2	5 (HELP)
	3	Type of help: 1 : command 2 : nonterminal
	4	Index in symtable of command or non-terminal

GET and LET

	1	Length of PT
	2	6 (GET) or 7 (LET)
	3	Length of PT used
	4	Index in symtable for label (0 if none)
	5	Operation type: 0 : none (single table) 1 : join 2 : union 3 : intersection 4 : difference 5 : append
Table expression (see below)		
	x	Order operator 0 : none 1 : up 2 : down
	x + 1	Descriptor number to order on

SAVE

	1	Length of PT
	2	8 (SAVE)
	3	Length of PT used
	4	Number of tables to be saved (ns)
	5	1st Table
	6	2nd Table
	.	
	.	
	.	
	4 + ns	Last table

TABLE EXPRESSION

	x	Number of tables
	x + 1	1st table
	x + 2	2nd table
	.	
	.	
	.	
	x + nt	Last table
	x + nt + 1	No. attributes (na)
	x + nt + 2	1st attribute
	x + nt + 3	2nd attribute

.
.
.

VL Condition (optional)

VL CONDITIONS

parsed as postfix codes, each code taking three places in PT:

```
x                          Length of this expression
                           (3n, n = number of codes)
x + 1
x + 2                      First code triple
x + 3
.
.                                                      Row
.          Operator       Attribute       Operand        Constant
x+3r-2        0               2              3               4
x+3r-1      opcode           RT#             0             Type
x+3r       #operands         DES#            0             Value
.
.
.
x + 3n - 2
x + 3n - 1                 Last code triple
x + 3n
```

Some examples of parsing of VL conditions:

```
:[row>10] or [P#=P1];
parsed as:
21  3 0 0   4 1 10   0 2 2   2 0 3   4 3 1   0 5 2   0 11 2
     row      10       >       P#      P1      =       OR

:[status < 2*qty];
parsed as:
15  2 0 9   4 1 2   2 0 8   0 9 2   0 1 3
    status    2      qty      *       <
```

REFERENCES

[1] Michalski, R. S., Baskin, A. B., et al.: "A Technical Description of the ADVISE Meta-expert System," Department of Computer Science, Internal Report, University of Illinois, Urbana, Illinois, 1983.

[2] Michalski, R. S., Davis, J. H., Bisht, V. S., and Sinclair, J. B., "PLANT/ds: An Expert Consulting System for the Diagnosis of Soybean Diseases," *1982 European Conference on Artificial Intelligence*, Orsay, France, July 12 - 14, 1982.

[3] Michalski, R. S., "A Theory and Methodology of Inductive Learning," Chapter in *Machine Learning: Artificial Intelligence Approach*, TIOGA Publishing Co., Palo Alto, R. S. Michalski, J. Carbonell and T. Mitchell (Eds.), 1983.

[4] Groner, G. F. et al., "An Introduction to the CLINFO Prototype Data Management and Analysis System, Release 2," Rand Corporation Report P-5617-1, February, 1977.

[5] Blum, R. L., "Discovery and Representation of Causal Relationships from a Large Time-oriented Clinical Database: The RX Project," Ph.D. Thesis, Stanford University, January 1982.

[6] Goldstein, L., "MEDUS/A: A High-level Database Management System," *Proceedings of the Fourth Annual Symposium on Computer Applications in Medical Care*, Washington, D. C., November 1980.

[7] Davis, R., "Interactive Transfer of Expertise: Acquisition of New Inference Rules," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977, pp. 321 - 328.

[8] Michalski, R. S., and Chilausky, R.L., "Learning by Being Told and Learning from Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis," *International Journal of Policy Analysis and Information Systems*, 1980, Vol. 4, No. 2, pp. 125-161.

[9] Michalski, R. S., "Variable-Valued Logic: System VL_1," *Proceedings of the 1974 International Symposium on Multiple-Valued Logic*, West Virginia University, Morgantown, West Virginia, May 29-31, pp. 323-346, 1974.

[10] Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* 13, No. 6, June 1970.

[11] Date, C. J., *An Introduction to Database Systems*, Addison-Wesley Publishing Company, 1981.

[12] Schubert, R. N., "The VL Relational Data Sublanguage for an Inferential Computer Consultant," Department of Computer Science, Report No. 846, University of Illinois, Urbana, Illinois, October 1977.

[13] Codd, E. F., "A Data Base Sublanguage Founded on the Relational Calculus," *Proceedings 1971 ACM SIGFIDET Workshop on Data Description, Access and Control.*

[14] Michalski, R. S., "Knowledge Acquisition Through Conceptual Clustering: A Theoretical Framework and an Algorithm for Partitioning Data into Conjunctive Concepts," A Special Issue on Knowledge Acquisition and Induction, *International Journal of Policy Analysis and Information Systems*, Vol. 4, No. 3, pp. 219 - 244, 1980.

[15] Stauffer, M., "GEM Users Manual," Department of Computer Science, Internal Report, University of Illinois, Urbana, Illinois, 1982.

[16] Baim, P. W., "The PROMISE Method for Selecting the Most Relevant Attributes for Inductive Learning Systems," Department of Computer Science, Internal Report ISG-82-1, University of Illinois, Urbana, Illinois, 1982.

[17] Cramm, S., "ESEL/2: A Program for Selecting the Most Representative Training Events for Inductive Learning," Department of Computer Science, Internal Report ISG-82-4, University of Illinois, Urbana, Illinois, 1982.

[18] McMillan, K., "NEWVAR: Constructing New Variables," Department of Computer Science, Internal Report, University of Illinois, Urbana, Illinois, 1982.

[19] Davis, J. H., "CONVART: A Program for Constructive Induction on Time Dependent Data," Master's Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1981.

[20] Michalski, R. S., and Larson, J. B., "Selection of Most Representative Training Examples and Incremental Generation of VL_1 Hypotheses : The Underlying Methodology and the Description of Programs ESEL and AQ11," Department of Computer Science, Report No. 867, University of Illinois, Urbana, Illinois, May 1978.

[21] Layman, T. C., "A PASCAL Program to Convert Extended Entry Decision Tables into Optimal Decision Trees," Department of Computer Science, Internal Report, University of Illinois, Urbana, Illinois, 1979.

[22] Michalski, R. S., Baskin, A. B., and Spackman, K. A., "A Logic-based Approach to Conceptual Database Analysis," *Sixth Annual Symposium on Computer Applications in Medical Care*, Washington, D. C., November 1982.

[23] Michalski, R. S., "Designing Extended-Entry Decision Tables and Optimal Decision Trees Using Decision Diagrams," Department of Computer Science Report No. 898, University of Illinois, Urbana, Illinois, March 1978.