PRELIMINARY DESIGN OF INGRES

PART I - QUERY LANGUAGE, DATA STORAGE AND ACCESS

by

Nancy McDonald, Michael Stonebraker and Eugene Wong

Memorandum No. ERL-M435

10 April 1974

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

PRELIMINARY DESIGN OF INGRES

by

Nancy McDonald, Michael Stonebraker and Eugene Wong

Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, California 94720

Part I - Query Language, Data Storage and Access

Part II - Protection, Concurrency, and Graphics

Abstract

INGRES (Interactive Graphics and Retrieval System) is designed to be a general purpose data base management system supporting a relational view of data. This report documents the query language QUEL (QUEry Language) available to the user, the data management features provided, the protection mechanism supported, the update concurrency allowed, and the available commands for the display of geographic oriented data.

Research supported in part by the National Science Foundation Grant GK-10656X3, the Army Research Office, Durham, Grant DAHCO4-74-G0087, the Joint Services Electronics Program Contract F44620-71-C-0087, the Naval Electronics System Command Grant N0039-71-C-0255 and the Sloan Foundation.

Forward

Project INGRES has its origin in two related projects. One was a computer-graphics system that had been developed by P. Macri and P. P. Varaiya to deal with map related data. The second was a workshop on data base systems conducted by M. R. Stonebraker and E. Wong, the objective of which was to design and implement a system incorporating certain land-use data of the City of San Francisco. In late 1973, certain common interests and goals became apparent to both groups, of which the foremost was the need for a hardware system. Thanks to financial support from the National Science Fouundation, the Sloan Foundation, the College of Engineering and the Department of Electrical Engineering and Computer Sciences at Berkeley, sufficient funds were obtained to purchase a PDP-11/40 based computer graphics system. It is expected that the first version of a graphics and retrieval system will be operational before the end of 1974. In this report we shall describe the basic design goals and philosophy as well as the preliminary system specifications.

Division of labor among the three listed co-authors was as follows: M.R.S. wrote sections 1, 4, 5, 6, 7 and Appendix C, E. W. wrote sections 2, 3 and appendices A and B, N. M. was responsible for section 8. P. P. Varaiya has been the principal moving spirit in the project from its inception. But for him, the project would not exist. Primary responsibility for implementing the retrieval portion of the system is borne by Karel Youssefi (Query Processor) and Peter Kreps (Access Processor). The graphics portion of the system is being implemented by Nancy McDonald and Barbara Cottrell. Members of the Database Workshop (EECS 298-15)

have implemented a preliminary version of the system commands given in Appendix C and have contributed ideas at every stage of the project.* We are also grateful to Peter Groat of the Planning Department of the City of San Francisco, to Dean E. S. Kuh of the College of Engineering and to Chairman T. E. Everhart of the Department of Electrical Engineering and Computer Sciences for their enthusiastic support.

*In addition to those already mentioned, the members include W. Chang, J.I. Chapela, J.M. Ford, P. Jahanian, P.G. Lehot, J.C. Liang, J.H. Liou, J.N. Yang.

1. Introduction

INGRES (INteractive Grapics and REtrieval System) is designed to be a general purpose data base management system. As such it provides facilities for controlled access to and efficient management of large data bases. In addition, specialized facilities for the display of geographic oriented data are provided.

1.1 Goals

The basic goals of the design are:

1. A design supporting a relational view of data.
2. A high level data definition, retrieval and update facility.
3. Data independence.
4. Sophisticated protection scheme
5. Efficient and flexible storage organization.
6. Flexible support of geographic oriented data.
7. Modular and extendable design.
8. A design implementable in one calendar year.

Goal 1 reflects our belief that a relational view of data [1,2,3] is the most appropriate one. In particular, it is preferable to a hierarchical view [4,5].

Goal 2 reflects our belief that interactions with a data base should be conducted in the highest level language possible. Examples of other high level languages include SEQUEL [6], VERS [7], Data Language/Alpha [8]. and Square [9]. Such languages typically free the programmer from concern for how data structures are implemented and what algorithms are operating on stored data. The language through which users interact with INGRES is called QUEL (QUEry Language) and is closely modelled after

the language ALPHA. The design of QUEL is strongly motivated by implementation considerations. Eventually we plan to design an even higher level language which will be compiled into QUEL, perhaps one approaching natural English [10].

Goal 3 results from our belief that programs in QUEL should not be rendered obsolete because of changes in the way data are physically sorted. Such changes may be desirable from time to time for efficiency reasons. Moreover, data bases tend to grow in the number of domains present. INGRES ensures that user programs will continue to operate correctly over a large class of storage transformations. For a further discussion of data independence the reader is referred to [11].

Goal 4 reflects our conviction that large data bases require access controls of a sophistication not found in existing systems [12]. One possible user of INGRES, the Dean of the College of Engineering (for storage and retrieval of student and faculty records) is particularly concerned about security.

The diverse and time-varying data bases that INGRES must manage require diverse storage organizations for efficient management. A fifth goal of INGRES is the support of a class of organizations that will efficiently accomodate most data bases.

Goal 6 reflects the fact that INGRES is an outgrowth of a far simpler system [13] now operational at Berkeley. The germination of this project resulted from user demands for a more powerful system.

Lastly, goals 7 and 8 reflect the pragmatic reality that support for the project has been generously provided by the College of Engineering and by a steering committee for interdisciplinary graduate education

through a grant from the Sloan Foundation under the assurance that an operational system should be available by December, 1974.

1.2 Environment

INGRES will run as a normal user job on top of the UNIX [14] operating system on a PDP-11/40 CPU [15]. In the sequel the reader is assumed to be familiar with the features of UNIX. The implementation of INGRES is being programmed in "C" [16] an extension of BCPL [17]. It was chosen as the most suitable language currently supported by UNIX.

Since our implementation of INGRES must be operational quickly, we have designed INGRES to require as few changes to "C" and UNIX as possible.

1.3 Report Organization

The remainder of this report is organized as follows. Section 2 and 3 indicate respectively the retrieval and update capabilities of QUEL and the mechanisms used to decompose interactions. Then in Section 4 the available methods for relation storage which are available in INGRES are presented. Sections 5 and 6 provide a discussion of the access controls and deadlock prevention mechanisms built into INGRES. In Section 7 the miscellaneous data management commands are described. Then, Section 8 documents the display oriented features built on top of the basic system. Lastly, Section 9 presents a summary and suggests conclusions and possible extensions.

2. Retrieval Capabilities in QUEL

The discussion commences with notational conventions adopted throughout this and subsequent sections. Then we indicate the format of QUEL retrieval statements.

2.1 Notation

Given sets $D_1,\ldots,D_n$ (not necessarily distinct) a relation $R(D_1,\ldots,D_n)$ is a subset of the Cartesian product $D_1 \times D_2 \times \ldots \times D_n$. In other words, R is a collection of n-tuples $x = (x_1,\ldots,x_n)$, where $x_i \in D_i$ for $1 \leq i \leq n$. The sets $D_1,\ldots,D_n$ are called the domains of R, and R has degree n.

The only restriction put on relations is that they be normalized [1]. Hence, every domain must be simple i.e. it cannot have members which are themselves relations.

Clearly R can be thought of as a table with elements of R appearing as rows and with columns labeled by domain names, as illustrated by the following example.

Example 2.1

Six tuples from a landuse relation with three domains, parcel number, parcel zoning, and parcel use are indicated

Landuse

PARCEL #	ZONE	USE
1	Residential	1-family
2	Industrial	Vacant
3	Commercial	Restaurant
4	Commercial	Mortuary
5	Residential	Apartment
6	Commercial	Go Go Club

Each column in a tabular representation of R can be thought of as a function mapping R into D_i. These functions will be called attributes. An attribute will not be separately designated but will be identified by names of the pair (relation, domain) defining it. Where no confusion can arise only the domain name need be given.

Throughout this section the above relation along with two other relations will be used for examples. These relations are:

Census

PAR #	UNIT #	# OCCU
1	1	5
4	1	2
5	1	2
5	2	4
5	3	3
5	4	6
6	1	4

and

City

C NAME	STATE	POPULATION
San Francisco	Ca.	700000
Berkeley	Ca.	100000
New York	N.Y.	8000000
St. Louis	Mo.	1500000

2.2 Format of RETRIEVE Queries

Queries to RETRIEVE are probably the most important ones in QUEL. Each RETRIEVE-query consists of: one RANGE declaration, an optional ALIAS declaration, an optional DESIGNATE declaration, and one or more RETRIEVE Statements.

a. RANGE declaration

Form - RANGE: Relation Name (Symbol,.....,Symbol):

Relation Name (Symbol,.....,Symbol):

⋮

Relation Name (Symbol,.....,Symbol)

Purpose - The symbols declared in the RANGE declaration are variables which will be used as arguments for attributes. These are called tuple variables. The purpose of the declaration is to specify the relation over which each variable ranges.

Scope - The scope of each RANGE declaration is limited to the current query.

Allowed Relations - The relations names allowed in a RANGE declaration are: those of the relations in the data base to which the current user has access, those of the relations that have been obtained as results of previous queries in the current sign-on session, those of the resultant relations of the current query provided that no ambiguity arises.

b. ALIAS declaration.

Form - ALIAS: Name (alias): Name (alias): ..... : Name (alias)

Purpose - To introduce abbreviated notations for domain names and relation names.

Scope - Current query.

c. DESIGNATE statement

Form - DESIGNATE: (Name = A-function): (Name = A-function): ..... (Name = A-function)

Purpose - The main purpose of DESIGNATE statement is to name any domain of the resultant relations of the current query which is not an existing domain, (for example, DENSITY = X. POPULATION/ X. AREA). While a secondary benefit of a DESIGNATE statement is abbreviation, it is a much less important one. Another situation which requires renaming arises when two domains

of a newly created relation have identical names.

Scope - The name assigned to any domain of a relation created by the current query has the life of that relation, viz., the current sign-on session unless the relation is made permanent by a separate procedure.

Default Condition: - If no name is assigned when one is needed, the newly created domain will be referred to as COLU i, according to its position in the newly created relation.

d. RETRIEVE statements

Form - RETRIEVE: Result Name: Target List: Qualification

Purpose - Each RETRIEVE statement creates a relation from one or more existing relations and assigns to it the name specified by "Result Name." Target List and Qualification will be explained in later sections.

Limitation - There is no limit on the number of RETRIEVE statements which can be included in a query. However, the RETRIEVE statements in a given query must be so ordered that no statement contains a tuple variable with a range which is named in a subsequent statement, i.e., all the ranges of a RETRIEVE statement must be well-defined as of that point.

Scope - The relation created by a RETRIEVE statement and its name survive for the current sign-on session, unless the relation is made permanent by a separate procedure.

Example 2.2 Consider two relations

LANDUSE (PARCEL#, ZONE, USE)

CENSUS (PARCEL#, UNIT#, # OCCUPANTS)

where the domain names appear within the parentheses

a. Find those parcels with at least one unit having more than 6 occupants.

RANGE: CENSUS (X)

RETRIEVE: W : X. PARCEL#: X.#OCCUPANTS > 6

b. Find all parcels zoned commercial which have occupants

RANGE: CENSUS(X): LANDUSE(Y)

RETRIEVE: W: X. PARCEL#: (X. #OCCUPANTS > 0)

$\wedge$ (X. PARCEL# = Y. PARCEL#)

$\wedge$ (Y. ZONE = C)

c. Find all pairs of distinct parcels which are identically zoned.

RANGE: LANDUSE(X,Y)

DESIGNATE: P1 = PARCEL#(X): P2 = PARCEL#(Y)

RETRIEVE: PAIR: P1,P2 : $(P_1 \neq P_2) \wedge$ (X.ZONE = Y. ZONE)

2.3 Attribute Function

If X is a declared tuple-variable with range R and D is the name of a domain in R, then the notation X.D. stands for the attribute defined by (D,R). Attribute functions are functions of declared tuple-variables constructed from attributes by simple mappings and by arithmetrical operation as defined below.

a. A constant is an attribute function.

b. An attribute is an attribute function.

c. If f and g are attribute functions then f^g and $\log_f g$ are attribute functions.

d. If f and g are attribute functions, not necessarily having identical arguments, then f + g, f - g, f * g, f/g are all attribute functions. The argument of the resultant function is the union of the variables in the arguments of f and g.

2.4 Aggregate-Free Qualification

Let CITY be a relation with domains CNAME, STATE, POPULATION. Let X and Y be tuple-variables with range CITY. The condition "X. POPULATION $\leq$ 1,000,000" specifies a subset of CITY, or equivalently a truth function on CITY. The condition "X. POPULATION > Y. POPULATION," on the other hand, specified a subset of CITY × CITY. For the condition

(X. POPULATION $\leq$ 1,000,000) and (X. POPULATION > Y. POPULATION)

to make sense the first subcondition (X. POPULATION $\leq$ 1,000,000) will be interpreted as also specifying a subset of CITY × CITY. The entire condition then nonambiguously specifies the intersection of two sets of CITY × CITY. These conditions typify the "Qualification" portion of a RETRIEVE statement. In this section we shall specify the allowed qualifications for RETRIEVE statements containing no aggregation operations. The necessary modifications to include aggregations are minor and will be given later.

a. Atomic Formulas

Let f and g be attribute functions, not necessarily having identical arguments, then f * g is an atomic formula for the following binary

operators:

$f * g \equiv f = g$

$f \geq g$

$f > g$

$f \geq g \geq 0$

$f > g \geq 0$

$f \geq g > 0$

$f > g > 0$

Note that a formula such as $f \geq g \geq h$ may appear to involve a ternary operator, but it can be reexpressed as $f - h \geq g - h \geq 0$. Negation of an atomic formula is again an atomic formula. ($\sim f * g$)

b. Clause

A clause is a conjunction (logical and) of a finite number of atomic formulas. The format for expressing a clause is given by

(atomic formula) $\wedge$ (atomic formula) $\wedge$ ..... $\wedge$ (atomic formula)

An atomic formula is a degenerate clause.

c. Sentence

A sentence is a disjunction (logical or) of a finite number of clauses to be expressed as: clause $\vee$ clause $\vee$ ... $\vee$ clause.
A clause is a degenerate sentence.

A proposition is either an atomic formula, or a clause, or a sentence. We shall say the preposition is atomic in the first case, conjunctive in the second, and disjunctive in the third. The "qualification" in a

RETRIEVE statement is either a proposition or void. Clearly, the qualification specifies a subset of the product of the ranges of the declared variables.

Example 2.3 If the range of both X and Y is CITY, then (X. POPULATION $\leq$ 1,000,000) $\wedge$ (X. POPULATION > Y. POPULATION) specifies the collection of pairs of tuples (X,Y) in CITY × CITY which satisfy both atomic formulas of the conjunctive proposition.

2.5 Aggregation

The following aggregation-operators are recognized in QUEL: COUNT, COUNT', SUM, SUM', AVG, AVG', MAX, MIN. These operators can be used in a number of different ways as explained below.

a. Unary Operators on a Clause

A clause CL (i.e., a conjunctive proposition) containing tuple variable, $X_1,X_2,\ldots,X_n$ can be viewed as a truth function $CL(X_1,X_2,\ldots,X_n)$ which is 1 if $(X_1,X_2,\ldots,X_n)$ satisfies the proposition and 0 otherwise. Suppose the range of $X = (X_1,X_2,\ldots,X_n)$ is $(R_1,R_2,\ldots,R_n)$. Then

$$COUNT(CL(X_1,X_2,\ldots,X_n))$$

= The number of points in $R_1 \times R_2 \times \ldots \times R_n$ such that $CL(\underline{X}) = 1$

COUNT is the only aggregation operator defined as a unary operator on a clause.

b. Unary Operators on an Attribute Function

Let $\underline{X} = (X_1,X_2,\ldots,X_n)$ be tuple variables with range $R_1 \times R_2 \times \ldots \times R_n$. Let $f(\underline{X})$ be an attribute function.

COUNT $(f(\underline{X}))$ = The number of points in $R_1 \times R_2 \times \ldots \times R_n$. (f is immaterial here.)

COUNT'$(f(\underline{X}))$ = The number of <u>different</u> values of $f(\underline{X})$ which occur as $\underline{X}$ ranges over $R_1 \times R_2 \times \ldots \times R_n$.

SUM$(f(\underline{X}))$ = The sum of all values of $f(\underline{X})$ as $\underline{X}$ ranges over $R_1 \times R_2 \times \ldots \times R_n$.

SUM'$(f(\underline{X}))$ = The sum of all <u>distinct</u> values of $f(\underline{X})$ as $\underline{X}$ ranges over $R_1 \times R_2 \times \ldots \times R_n$. (Duplicate values are eliminated before summing.)

AVG$(f(\underline{X}))$ = SUM$(f(\underline{X}))$/COUNT$(f(\underline{X}))$

AVG'$(f(\underline{X}))$ = SUM'$(f(\underline{X}))$/COUNT'$(f(\underline{X}))$

MAX$(f(\underline{X})$ = The largest value of $f(\underline{X})$ for $\underline{X} \in R_1 \times \ldots \times R_n$.

MIN$(f(\underline{X}))$ = The smallest value of $f(\underline{X})$ for $\underline{X} \in R_1 \times \ldots \times R_n$.

<u>Note</u>: 1. One gets garbage for SUM, SUM', AVG, AVG' if f is not numerical-valued. At some later time more extensive type checking may be introduced.

2. One gets nonsense for MAX and MIN if the values of f are not ordered.

3. For any aggregation-operator Σ, $\Sigma(f(\underline{X}))$ is a number and not a function. The presence of $\underline{X}$ merely indicates the range.

<u>c. Binary Operators on (Attribute Function; Clause)</u>

Suppose $f(\underline{X})$ is an attribute function and $CL(\underline{X})$ is a clause with identical arguments. Let S be the set in $R_1 \times \ldots \times R_n$ for which $CL(\underline{X}) = 1$. Then, for any aggregation-operator Σ, $\Sigma(f(\underline{X}); CL(\underline{X}))$ has the same meaning as $\Sigma(f(X))$, except that S replaces $R_1 \times \ldots \times R_n$ in its definition.

If the attribute function and the clause do not have identical arguments, say $f(\underline{Y})$ and $CL(\underline{Z})$, then $\Sigma(f(\underline{Y}); CL(\underline{Z}))$ is interpreted as $\Sigma(\bar{f}(\underline{X}); \overline{CL}(\underline{X}))$ where $\underline{X}$ is the union of the variables in $\underline{Y}$ and $\underline{Z}$, and $\bar{f}$ $\overline{CL}$ are extensions of f and CL to $\underline{X}$ obtained by treating them as being constant on the added variables. Some care must be exercised in using aggregation in this way, since it has consequences which may not be obvious.

Note: $\Sigma(f(\underline{X}); CL(\underline{X}))$ or $\Sigma(f(\underline{Y}); CL(\underline{Z}))$ is a constant not a function. The variables are dummy variables used to indicate range.

d. Binary Operators on (Attribute Function; Attribute Function)

Let $f(\underline{X})$ and $g(\underline{X})$ be attribute functions with identical arguments. Let the range of $\underline{X}$ be $R_1 \times R_2 \times \ldots \times R_n$. The distinct values of g partition $R_1 \times R_2 \times \ldots \times R_n$ into subsets of the form

$$S_\alpha = \{\underline{X}: g(\underline{X}) = \alpha\}$$

For any aggregation-operator Σ=(COUNT, COUNT', SUM, SUM', AVG, AVG', MAX, MIN) we define $\Sigma(f(\underline{X}); g(\underline{X}))$ as follows:

(i) $\Sigma(f(\underline{X}); g(\underline{X}))$ is a function on $R_1 \times R_2 \times \ldots \times R_n$

(ii) $\Sigma(f(\underline{X}); g(\underline{X}))$ is constant on each set S_α and

$$\Sigma(f(\underline{X}); g(\underline{X}))\Big|_{\underline{X}\in S_\alpha} = \Sigma(f(\underline{X}); g(\underline{X}) = \alpha)$$

Note: (1) In the defining equation in (ii) the right hand side is of the form Σ(attribute function; clause).

(2) Unlike previous cases, $\Sigma(f(\underline{X}); g(\underline{X}))$ is a function of the variables $\underline{X}$.

(3) Since $\Sigma(f(\underline{X}); g(\underline{X}))$ is constant on any set of $\underline{X}$ on which $g(\underline{X})$ is constant, it is a function of $g(\underline{X})$.

Functions of the form $\Sigma(f(\underline{X}); g(\underline{X}))$ will be called aggregate-functions. On the other hand Σ(clause), Σ(attribute function), and Σ(attribute function, clause) are all constants and will be called aggregates.

If f and g have different arguments, say $\underline{Y}$ and $\underline{Z}$, then $\Sigma(f(\underline{Y}); g(\underline{Z}))$ will be interpreted as $\Sigma(\bar{f}(\underline{X}); \bar{g}(\underline{X}))$, where $\underline{X}$ is the union of variables in $\underline{Y}$ and $\underline{Z}$, $\bar{f}$ and $\bar{g}$ are extensions of f and g to $\underline{X}$. Again, we call attention to the fact that $\Sigma(f(\underline{Y}); g(\underline{Z}))$ may have non-obvious implications and must be used with some care.

Let $\underline{g} = (g_1, g_2, \ldots, g_m)$. It is easy to see how $\Sigma(f(\underline{X}); \underline{g}(\underline{X})) = \Sigma(f(\underline{X}); g_1(\underline{X}), g_2(\underline{X}), \ldots, g_m(\underline{X}))$ can be defined. It is constant on any set of $\underline{X}$ for which $\underline{g}(\underline{X})$ is constant, and

$$\Sigma(f(\underline{X}); \underline{g}(\underline{X}))\Big|_{\underline{g}(\underline{X}) = \underline{\alpha}} = \Sigma(f(X); \underline{g}(X) = \underline{\alpha})$$

Examples to follow will make use of the two relations CENSUS and LANDUSE given in section 2.1.

Range: LANDUSE(X): CENSUS(Y)

Ex. 2.4 Count the number of parcels in LANDUSE zoned commercial

COUNT (X. ZONE = C)

Ex. 2.5 Count the total number of units in CENSUS

COUNT (Y. UNIT#)

Ex. 2.6 Count the number of different parcels included in CENSUS

COUNT'(Y. PAR#)

Ex. 2.7 Find the total number of occupants in CENSUS

SUM(Y. #OCCU)

Ex. 2.8 Find the average number of occupants per unit in CENSUS

AVG(Y. #OCCU)

Ex. 2.9 Find the average number of occupants per unit in parcels zoned residential.

AVG(Y. #OCCU; (Y. PAR# = X. PAR#) ^ (X. ZONE = R))

Ex. 2.10 Find the total number of occupants in parcels zoned commercial.

SUM(Y. #OCCU; (Y. PAR# = X. PAR#) ^ (X. ZONE = C))

Ex. 2.11 For each parcel find the total number of units.

COUNT(Y. UNIT#; Y. PAR#)

In QUEL nested aggregations are not allowed in the same statement. For example, the statement

COUNT(Y. OCCU > AVG(Y. OCCU))

makes sense, but is illegal in QUEL. In order to perform such an operator, we need two statements. This can be done in various ways. One example is the following:

```
RANGE:  CENSUS(Y): MEAN(Z)
DESIGNATE: AV = AVG(Y. OCCU)
RETRIEVE: MEAN: AV
RETRIEVE: W: COUNT(Y. OCCU > Z. AV)
```

Another example of achieving the same thing using legal QUEL statements is given below.

```
RANGE: CENSUS(Y): CROWD(Z)
RETRIEVE: CROWD:  Y. PAR#, Y. UNIT#, Y. #OCCU:
                  Y. #OCCU > AVG(Y. #OCCU)
```

Ex. 2.12

RETRIEVE: W: COUNT(Z.#OCCU)

As a further example involving nested aggregation, consider the query: "Find the average number of occupants per parcel in CENSUS." The following QUEL query does the job.

Ex. 2.13

RANGE: CENSUS(X): TEMP(Y)

DESIGNATE: POP = SUM(X.#OCCU; X. PAR#)

RETRIEVE: TEMP: X. PAR#, POP

RETRIEVE: W: AVG(Y. POP)

2.6 A - Functions

We define a scalar to be a constant, or an aggregate, or any arithmetical combination of constants and aggregates. We define an A-function as follows:

(a) A scalar is an A-function

(b) An attribute function is an A-function

(c) An aggregate function is an A-function

(d) The class of A-functions is closed under the mappings:

f^g, $\log_f g$ (f,g = A-functions)

(e) The class of A-functions is closed under binary operations:

+, -, *, /.

2.7 Qualifications

To modify the definitions given in section 2.4, we only need to modify the atomic formulas: If f and g are A-functions, then f*g is an atomic formula for all the binary operators * specified in (2.4a). Conversely, any atomic formula must be of this form.

As before, a clause is a conjunction of atomic formulas and a

sentence is a disjunction of clauses. Collectively, atomic formulas, clauses, and sentences will be referred to as propositions. The qualification portion of a retrieve statement is either void, or is a proposition.

We note that clauses serving as operands in aggregates can involve only attribute functions.

2.8 Target List

The target list of a RETRIEVE statement has the form

A-function , A-function , ... , A-function

where the A-functions may appear either as formulas defining them, or as names defined in a DESIGNATE statement.

Suppose X_1, X_2, ..., X_n are the declared tuple variables, with ranges R_1, R_2, ..., R_n. Then the "Qualification" defines a subset S of $R_1 \times R_2 \times \ldots \times R_n$. Each A-function in the target list can be viewed as a function on $R_1 \times R_2 \times \ldots \times R_n$, whether or not all the tuple variables appear explicitly in the function. We now interpret the resultant relation of the RETRIEVE statement as being constructed by:

(a) evaluating the A-functions in the target list on S in the order specified, and

(b) ordering and eliminating duplicate tuples resulting from (a)

A formal description of the syntax of the retrieval queries is given in Appendix A. A query parsing algorithm is indicated in Appendix B.

2.9 Special Keywords and Conventions

a. ALL

Let X, Y be tuple variables with range R and let D_1, D_2, ..., D_n be the domains of R. When used in the target list X. ALL means $X.D_1$, $X.D_2$,...,$X.D_n$. When used in the qualification X.ALL = Y.ALL means $(X.D_1 = Y.D_1) \wedge (X.D_2 = Y.D_2) \wedge \ldots \wedge (X.D_n = Y.D_n)$.

b. ID

The keyword ID is the name for a tuple-enumerator. In QUEL all relations, unless otherwise specified, are assumed to be sorted in the order of the domains. X.ID is equal to i for the i^{th} tuple.

c. COUNT(R)

A count of the number of tuples is maintained for every existing relation, and every newly created relation. If R is the name of a relation, COUNT(R) yields the number of tuples in R.

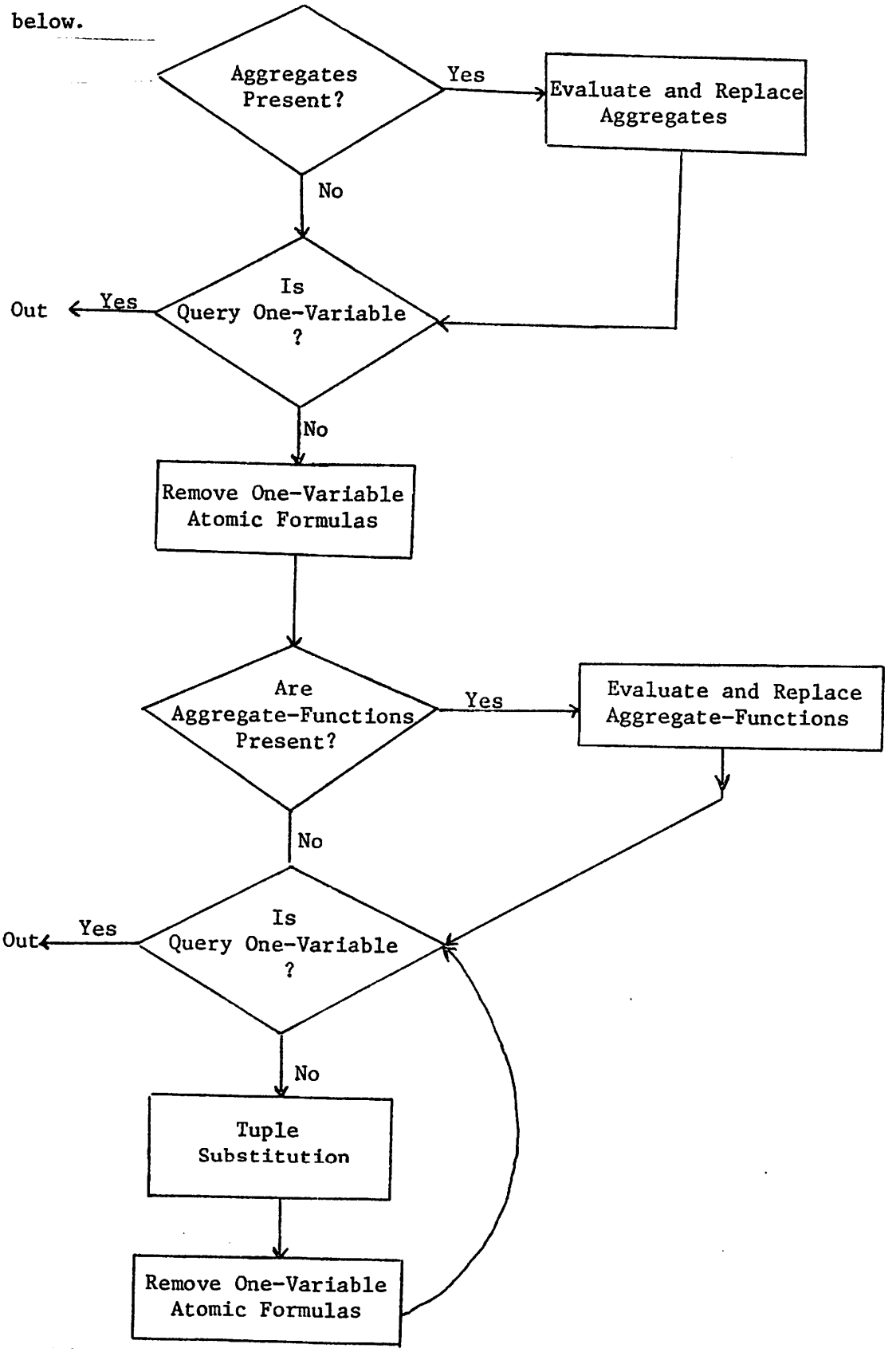
2.10 Decomposition

The processing of a RETRIEVE query begins by removing the disjunctions and repeating the query for each clause. The results will then be merged. Basically, the processor prefers to deal with queries containing only conjunctive propositions (i.e. clauses) in their qualification. The next step in the processing is to replace each query by a sequence of 1-variable queries. This decomposition is a vital part of the system.

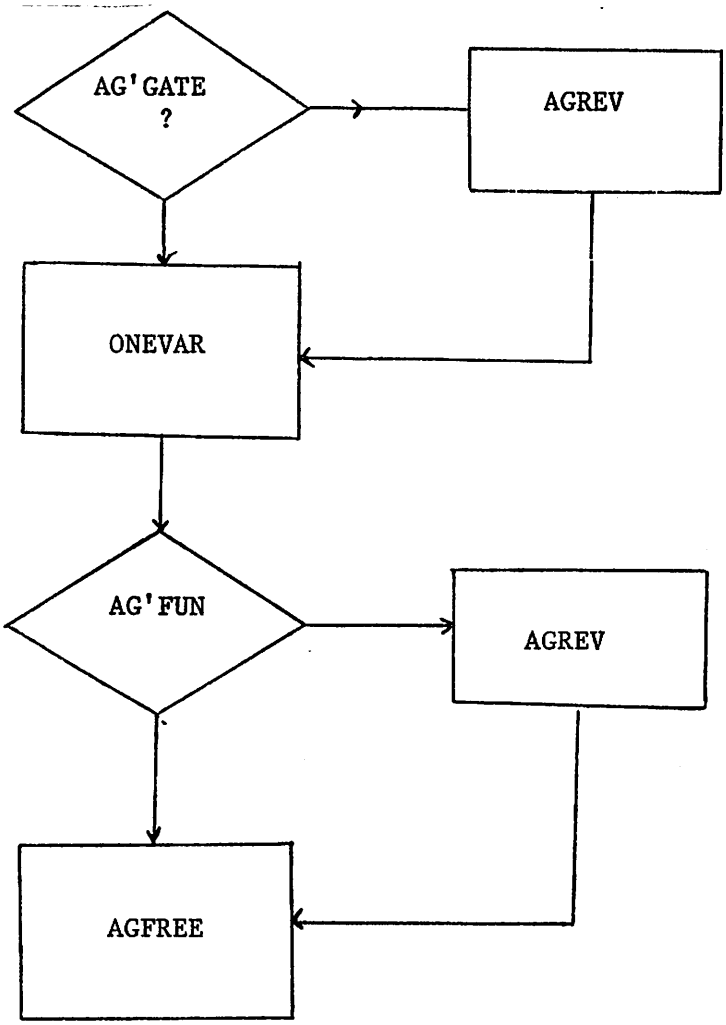
The most difficult problem in the decomposition is to deal with aggregates and aggregate-functions. Several ways of solving their problem were considered. The particular procedure adopted was chosen on the basis of programming simplicity and may not be the most efficient in terms of overall execution speed. The basic idea in the approach is to replace aggregate-functions by attribute functions defined on newly

created relations.

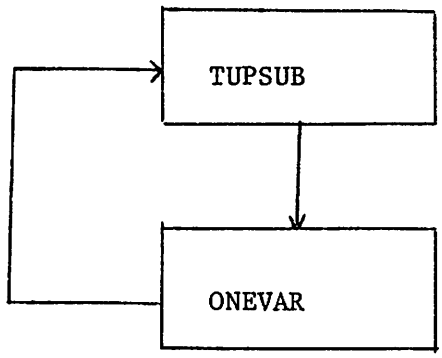
The principal components in the decomposition procedure are illustrated below.



Grouping some of the components makes the sub algorithms a little more evident.



The subprocess AGFREE in turn is represented by



The details of the component processes are explained below.

a. ONEVAR

(i) The query being presented is examined to determine whether it involves only one variable. If so, the sub-procedure ends. If not, go on.

(ii) For each variable X determine C(X) = conjunction of all atomic formulas involving only X. If C(X) is void for every X, the sub-procedure ends, otherwise go on.

(iii) Let R be the range of X. Determine $X.A_1, X.A_2, \ldots, X.A_k$, the attributes involving X appearing in the remainder of the query (target list or qualification) after C(X) is deleted.

(iv) Issue the query

RANGE: R(X)

RETRIEVE: R': $X.A_1, X.A_2, \ldots, X.A_k$: C(X)

(v) Delete C(X) from the qualification and change R to R' in the original query

End

b. TUPSUB (Tuple Substitution)

Note: the query being presented can be assumed to be free of aggregation operators, to contain no one-variable formula, to involve at least two variables.

(i) Determine X, the tuple variable with a range say R having the least count among all ranges of the query.

(ii)) Determine the attributes in the query which depend on X, say $X.A_1, X.A_2, \ldots, X.A_m$.

(iii) For $1 \leq i \leq$ COUNT(R), issue the query

RANGE: R(X)

RETRIEVE: W_i: $X.A_1, X.A_2, \ldots, X.A_m$: X.ID = i

(iv) A special arrangement is made to return W_i by value, and the values so returned are substituted into the query for $X.A_1, X.A_2, \ldots, X.A_m$.

(v) X is deleted as a variable in the query

c. AGREV (for aggregates)

Any aggregate can be assumed to be of the form

Σ(attribute function; clause)

because COUNT(clause) = COUNT(1;clause) and Σ(Attribute function) = Σ(attribute function; Void). Note also that the variables in an aggregate are dummy variables used only to indicate range. Whether the same variables appear elsewhere or not is immaterial.

(i) If the aggregate involves only one variable, say X, then a one-variable query is issued, the value returned as substituted into the original query. If X appears nowhere else in the query, it is eliminated.

(ii) If the aggregate involves several variables, say $\underline{X} = X_1, \ldots, X_m$, then the aggregate must be of the form $\Sigma(f(\underline{X}.\underline{A}); CL(\underline{X}))$, where $\underline{A}$ are domain names and CL is a clause. Denote the ranges of $\underline{X}$ by $R_1, R_2, \ldots, R_m$. Issue the queries

Q_1

RANGE: $R_1(X_1) : R_2(X_2) : \ldots : R_m(X_m)$

RETRIEVE: V: $\underline{X}.\underline{A}$: CL($\underline{X}$)

Q_2

RANGE: V(Y)

RETRIEVE: W: Σf(Y.A))

(iii) Observe that Q_1 is a multiple-variable query but free of aggregation operators, and hence can be decomposed by AGFREE. On the other hand Q_2 is a one-variable query.

(iv) The value returned is substituted in the original query and components of $\underline{X}$ not appearing elsewhere in the original query are eliminated.

d. AGREV (For Aggregation-Functions)

Unlike aggregates, the variables in an aggregate-function are not dummy variables.

(i) If the aggregate function involves only a single variable, say X with range R, then it must be of the form

$$\Sigma(f(X.\underline{A});\ g(X.\underline{B}))$$

where $\underline{A}$ and $\underline{B}$ are domain names. Issue a query

RANGE: X(R)

DESIGNATE: C = Σ(f(X.$\underline{A}$; g(X.$\underline{B}$)):

D = g(X.$\underline{B}$)

RETRIEVE: W: D,C

If X appears nowhere else in the original query, change its range from R to W and replace the aggregate-function by X.C. If X does appear somewhere else in the original query, add W(Z) to its range declaration, replace the aggregate-function by Z.C and add the atomic formula g(X.B) = Z.D to the qualification.

(ii) If the aggregate function involves several variables, say $\underline{X} = X_1, X_2, \ldots, X_m$, with ranges $R_1, R_2, \ldots, R_m$, it must be of the form

$\Sigma(f(\underline{X}.\underline{A}); \underline{g}(\underline{X}.B))$. Issue two queries

Q_1
RANGE: $R_1(X_1)$: $R_2(X_2)$: ...: $R_m(X_m)$

RETRIEVE: V: C,$\underline{D}$

Q_2
RANGE: V(Z)

DESIGNATE: E = $\Sigma(f(\underline{X}.\underline{A}); \underline{g}(\underline{X}.\underline{B}))$

RETRIEVE: W: Z.$\underline{D}$,E

Note that Q_1 is multi-variable but aggregation-free, and can be decomposed by using AGFREE. In the original query if $\underline{X}$ appears nowhere else, we replace $\underline{X}$ by Z with range W and $\Sigma(f(\underline{X}.\underline{A}); \underline{g}(\underline{X}.\underline{B}))$ by Z.E. If one or more components of $\underline{X}$ appears elsewhere in the original query, we add W(Z) to the range declaration, replace the aggregate-function by Z.E. and add $\underline{g}(\underline{X}.B) = Z.\underline{D}$ to the qualifications.

Next, we turn to a consideration of the three forms of update statements.

3. Update Statements

In addition to RETRIEVE, QUEL permits three operations: REPLACE, DELETE and COMBINE, which are basically update operations. The meaning of REPLACE and DELETE are obvious, while COMBINE is a generalization of the operation usually called "insert." In one of its simplest applications, COMBINE is used to form the union of relations with identical domains.

3.1 DELETE

RANGE DECLARATION

DELETE: RESULT NAME: TARGET LIST: QUALIFICATION

The target list in a DELETE statement is restricted to contain only attributes of a single declared tuple variable and X, i.e.,

X. Domain Name, X.Domain Name, ..., X.Domain Name

The interpretation of the operation is as follows:

(i) Take the relation which is the range of X and delete from it the tuples which satisfy the QUALIFICATION.

(ii) Project the result of (i) on the domains named in the TARGET LIST.

Note: One can use the convention X.ALL as the TARGET LIST, if only deletion and not projection is desired.

Example 3.1

Delete from LANDUSE those parcels which appear in CENSUS to have multiple units.

RANGE: LANDUSE(X): CENSUS(Y)

DELETE: LANDUSE': X.ALL:

(X.PAR# = Y.PAR#)

$\wedge$ (COUNT'(Y.UNIT#; Y.PAR#) > 1)

3.2 REPLACE

RANGE DECLARATION

REPLACE: RESULT NAME: TARGET LIST: QUALIFICATION

The TARGET LIST of a REPLACE statement has the form

X.Domain Name = A-Function, X.Domain Name = A-Function, ...,

X.Domain Name = A-Function

where X must be the same variable in each term. The A-functions can be functions of declared variables other than X, and of attributes of X which do not appear on the left hand side of any term. On the set

S specified by the QUALIFICATION, each term in the TARGET LIST must represent a single-valued correspondence. That is, on any subset of S such that the attribute specified by the left hand side of a term is constant the A-function on the right hand side must be a constant.

(i) In the relation say R , which is the range of X, determine the sub-relation which satisfies the QUALIFICATION.

(ii) Replace on the sub-relation found in (i) the attribute values as specified by the TARGET LIST.

(iii) The resultant relation is R modified according to (ii).

EXAMPLE 3.2

Replace in LANDUSE the zoning classification of any multiple-unit parcel by M.

RANGE: LANDUSE(X): CENSUS(Y)

REPLACE: LANDUSE': X.ZONE = M:

(X.PAR# = Y.PAR#)

$\wedge$ (ROUNT'(Y.UNIT#: Y.PAR#) > 1)

3.3 COMBINE

RANGE DECLARATION

COMBINE: RESULT NAME: TARGET LIST: QUALIFICATION

Again only the TARGET LIST is different from RETRIEVE statements. The TARGET LIST in a COMBINE statement has the form

(Domain Name, ...,Domain Name)(tuple variable,.....,tuple variable)

The range of each of the tuple variables must contain every domain named in the TARGET LIST. We interpret a COMBINE statement as follows:

(i) For each variable named in the TARGET LIST, form a relation with domains as specified and using the QUALIFICATION

(ii) Form a union of all the relations obtained in (i) with tuples ordered.

EXAMPLE 3.3 Find the set of all parcels which are either zoned industrial or has more than one unit.

```
RANGE: LANDUSE(X): CENSUS(Y)
COMBINE: P: PAR#(X,Y):
            (X.ZONE = I)
         ∧ (COUNT'(Y.UNIT#; Y.PAR#) > 1)
```

4. Relation Storage in INGRES

All entities to be physically stored by INGRES are relations. In all cases a relation is stored in a single UNIX file and will be either permanent or temporary (workspaces). Each relation is given a unique identifier which is also the file name in which it is stored. Three things are noteworthy concerning relation storage

a) the access methods supported

b) the handling of variable length values

c) the use of indexes

4.1 Access Methods

At the moment five different means of storing a relation are available. The user need not be concerned about which one is currently in use.

The methods supported are

a) hash tables [18,19]

b) sorted tables

c) unsorted tables

d) compressed sorted tables

e) lists

Each storage structure has an associated access method and all access methods accept the same set of nine commands from higher level software, except that command 5 cannot make sense with hash access. New access methods can be easily added by implementing these commands for a new storage structure. This set of commands is as follows

1. Return the tuple from a specified relation offset from the last tuple returned by a positive or negative integer constant. (Each access method must include an ordering of its tuples to allow such requests. For sorted tables and lists it must be the obvious ordering.)

2. Return the tuple from a specified relation with the smallest key value greater than or equal to an offered key. This key is (an arbitrary number) K bytes long and is compared against the leftmost K bytes of tuples in the relation.

3. Return the first tuple that exactly matches an offered key. Again a key is the leftmost K bytes of a tuple.

4. Reset (i.e. set the last tuple returned to the zeroeth tuple).

5. Insert an offered tuple at the end of a specified relation.

6. Insert an offered tuple at its correct position in a specified relation.

7. In a specified relation delete all tuples with a key equal to an offered key.

8. Delete the last tuple retrieved

We now discuss the storage structure supported by each access method.

a) Hash access

We are implementing a straightforward hashing method utilizing division as a randomizing algorithm [20, 21]. Collisions will be resolved by rehashing to a new address. The hash table will be expanded when nearly full by selecting a larger modulus and rehashing the table. Since the number of tuples in a relation is expected to change slowly a more sophisticated scheme(for example [22]) does not appear warranted.

b & c) Sorted and unsorted tables

These are implemented in the obvious way.

d) Compressed tables

A sorted table may be compressed to save storage space if a severe penalty in access time is tolerable. Basically, the compression algorithm makes use of the fact that a tuple differs from its successor only in a certain number of right hand bytes, and only the difference need be stored for unique decipherability. Since the length of each tuple stored must also be recorded, expansion rather than compression results if the leftmost byte of a tuple never matches its predecessor. The compression scheme is most advantageous if the domains of a relation are arranged so that those with a small number of possible values are to the left.

e) Lists

The current implementation is a simple two-way linked list [23]. A structure similar to a B-tree [24] may be used in its stead in the future.

4.2 Handling Variable-Length Values

It was decided for programming ease that the above access methods would deal only with tuples of a fixed length. Thus, domains (such as person - names) which are typically of variable length character strings must be made fixed-length. Two schemes are allowed in INGRES:

1) padding with blanks
2) coding.

When there is a large variance in length, the first scheme is space-wasting. Hence, a coding routine is also provided. This routine accepts the following five commands.

a) Return the fixed length code corresponding to a specified variable length data item.
b) Return the data item corresponding to a specified code.
c) Insert a given data item and return a new code for it.
c) Delete the code offered and its associated data item.
e) Delete the data item offered and its associated code.

Since conversion between codes and data items must go in both directions, a coding tree is an appropriate mechanism. Conversion in each direction occurs in a time proportional to log n where n is the number of data items in the tree. For programming ease we are implementing a binary tree similar to the one in MacAIMS [25].

Schemes exist which ensure that binary trees do not become excessively unbalanced [26, 27, 28]. Unfortunately, balancing a coding tree requires the codes to be changed, a time-consuming process. Therefore, the decision was made to balance the tree dynamically only when one is forced to do so. This will happen when the insertion of a new data item causes the maximum depth of the tree to be exceeded. For convenience, an off-line utility will be provided to periodically inspect trees for poor balance and to correct them.

A decision has yet to be made as to how many coding trees will be maintained. In [25] all character strings are in one tree. At the other extreme each coded domain can have its own tree.

The latter situation is undesirable since a comparison of values for two different coded domains then requires the values to be decoded. The former situation will create one very large tree which will of necessity be very deep. Hence decoding, will be costly.

We shall adopt an intermediate position whereby groups of domains which are often compared will be coded in the same tree. Some domains, however, will have an individual tree. These include coded domains in indices.

4.3 Indices

An index can be created for any group of one or more domains in any relation. Although mechanisms exist for making good choices [29, 30], no attempt will be made to automate the selection process. INGRES will depend on commands from the "owner" of a data base to create indices. An example of indexing is now provided.

Example 4.1

An employee relation with four tuples is indicated below together with an index for one of the domains

	Name	Salary	Dept.	
EMP	SMITH	1000	English	tuple 1
	JONES	1500	Chemistry	tuple 2
	ADAMS	1000	Biology	tuple 3
	EVANS	1500	Engineering	tuple 4

Salary	Pointer
1000	1,3
1500	2,4

The pointer domain is variable-length so it is coded into an individual coding tree. The index relation is then stored and accessed just like any other relation. Normally, however, a sorted table is the expected storage choice for indices.

Groups of domains can be indexed as the following index relation indicates.

Sal	Dept	Pointer
1000	Bio	3
1000	English	1
1500	Chem.	2
1000	Eng.	4

When INGRES responds to a one-variable query, it attempts to use any redundant indices that will speed access.

References

[1] Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," CACM, Vol. 13, No. 6, (June 1970).

[2] Codd, E. F., "Relational Completeness of Data Base Sublanguages," Courant Computer Science Symposium 6, May, 1972.

[3] Codd, E. F., "Normalized Data Base Structure: A Brief Tutorial," Proc. 1971 ACM-SIGFIDET Workshop on Data Description Access and Control, San Diego, Ca., Nov. 1971.

[4] Information Management System/360, Program Description IBM Corp., SH20-0634.

[5] Information Management System/360, General Information Manual IBM Corp., GH20-0765

[6] Chamberlin, D. and Boyce, R., "SEQUEL A Structured English Query Language," IBM Research, San Jose, Ca.

[7] Early, J., "High Level Operations in Automatic Programming," Dept. of Computer Science, Univ. of California, Berkeley, Tech. Report 22, October 1973.

[8] Codd, E. F., "A Data Base Sublanguage Founded on the Relational Calculus," Proc. 1971 ACM-SIGFIDET Conference on Data Description Access and Control, San Diego, Ca., Nov. 1971.

[9] Boyce, R. F. et al., "Specifying Queries as Relational Expressions: SQUARE," IBM Research, San Jose, Ca., RJ 1291.

[10] Codd, E. F., "Seven Steps to Rendevous with the Casual User," IBM Research, San Jose, Ca. RJ 1333, January, 1974.

[11] Stonebraker, M., "A Functional View of Data Independence," submitted for publication.

[12] Feature "Analysis of Generalized Data Base Management Systems," Codasyl Systems Committee, May, 1971.

[13] Macri, P., "BUDS: Berkeley Urban Data System," Electronics Research Laboratory, Univ. of California, Berkeley, Technical Memorandum M412, Nov., 1973.

[14] Ritchie, E. and Thompson, K., "The UNIX Time-Sharing System," Proc. 6th Operating Systems Symposium, Yorktown Heights, N. Y., Nov. 1973.

[15] PDP 11/40 Processor Handbook, Digital Equipment Corp.,1972.

[16] C Reference Manual (unpublished)

[17] Richards, M., "BCPL: A Tool for Compiler Writing and System Programming," Proc. AFIPS SJCC, 1969.

[18] Maurer, W. D., "An Improved Hash Code for Scatter Storage," CACM Vol. 11, No. 1, January, 1968.

[19] Morris, R., "Scatter Techniques," CACM, Vol. 11, No. 1, January, 1968.

[20] Lum, V. "General Performance Analysis of Key-to-Address Transformation Methods Using an Abstract File Concept," CACM, 16, No. 10, October 1973.

[21] Lum, V. et al., "Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files," CACM, 14, No. 4, April 1971.

[22] Knott, G. D., "Expandable Open Addressing Hash Table Storage and Retrieval," Proc. 1971 ACM-SIGFIDET Workshop on Data Description Access and Control, San Diego, Ca., Nov. 1971.

[23] Knuth, D. E., The Art of Computer Programming, Vol. 1, Addison-Wesley, Reading, Mass., 1968.

[24] Bayer, R. and Mc Creight, E. M., "Organization and Maintenance of Large Ordered Indices," Proc. 1970 ACM-SIGFIDET Workshop on Data Description Access and Control, Houston, Texas, Nov. 1970.

[25] Goldstein, R. C. and Strnad, A. L., "The MacAIMS Data Management System," Proc. 1970 ACM-SIGFIDET Workshop on Data Description Access and Control, Houston, Texas, Nov., 1970.

[26] Adelson-Velskii, G. M. and Landis, E. M., An Information Organization Algorithm, DANSSSR, No. 2, 1962.

[27] Nievergelt, J., "Binary Search Trees and File Organization," Proc. 1972 SIGFIDET Workshop on Data Description Access and Control, Denver, Colorado, Nov. 1972.

[28] Walker, W. A. and Gotlieb, C. C., "Hybrid Trees: A Data Structure for Lists of Keys," Proc. 1972 ACM SIGFIDET Workshop on Data Description Access and Control, Denver, Colo., Nov. 1972.

[29] Stonebraker, M., "Retrieval Efficiency Using Confined Indices," Proc. 1972 ACM-SIGFIDET Workshop on Data Description Access and Control, Denver, Colo., Nov. 1972.

[30] Stonebraker, M., "The Choice of Partial Inversions and Combined Indices," Journal of Information and Computer Science, (to appear).

APPENDIX A - SYNTAX

RETRIEVE QUERY	=	DECLARATION
		RETRIEVE SEQUENCE
DECLARATION	=	RANGE DECLARATION
	=	RANGE DECLARATION
		ALIAS DECLARATION
	=	RANGE DECLARATION
		DESIGNATE DECLARATION
	=	RANGE DECLARATION
		ALIAS DECLARATION
		DESIGNATE DECLARATION
RETRIEVE SEQUENCE	=	RETRIEVE STATEMENT
	=	RETRIEVE STATEMENT
		RETRIEVE SEQUENCE
RANGE DECLARATION	=	RANGE: RELATION-NAME(VARIABLES):
		.....: RELATION-NAME(VARIABLES)
ALIAS DECLARATION	=	ALIAS: NAME(ALIAS): .....
		.....: NAME(ALIAS)
DESIGNATE DECLARATION	=	DESIGNATE: NAME=A-FUNCTION:
		.....: NAME = A-FUNCTION
RETRIEVE STATEMENT	=	RETRIEVE: RESULT-NAME: TARGET LIST:
		QUALIFICATION
TARGET LIST	=	A-FUNCTION
	=	NAME
	=	A-FUNCTION, TARGET LIST, NAME, TARGET LIST

QUALIFICATION	=	VOID
	=	PROPOSITION
PROPOSITION	=	CLAUSE
	=	CLAUSE $\vee$ PROPOSITION
CLAUSE	=	ATOMIC FORMULA
	=	ATOMIC FORMULA $\wedge$ CLAUSE
ATOMIC FORMULA	=	(A-FUNCTION * A-FUNCTION)
	=	F(A-FUNCTION, A-FUNCTION)
	=	~ ATOMIC FORMULA
*	=	=
	=	>
	=	$\geq$
F(f,g)	=	$f > g > 0$
	=	$f \geq g > 0$
	=	$f > g \geq 0$
	=	$f \geq g \geq 0$
A-FUNCTION	=	SCALAR
	=	ATTRIBUTE FUNCTION
	=	AGGREGATE FUNCTION
	=	M(A-FUNCTION,A-FUNCTION)
	=	A-FUNCTION $ A-FUNCTION
M(f,g)	=	f^g
	=	$\log_f g$

	=	+
	=	-
	=	*
	=	/
SCALAR	=	CONSTANT
	=	AGGREGATE
ATTRIBUTE FUNCTION	=	CONSTANT
	=	ATTRIBUTE
	=	M (ATTRIBUTE FUNCTION,ATTRIBUTE FUNCTION)
	=	ATTRIBUTE FUNCTION $ ATTRIBUTE FUNCTION
AGGREGATE FUNCTION	=	Σ(ATTRIBUTE FUNCTION;ATTRIBUTE SEQUENCE)
ATTRIBUTE SEQUENCE	=	ATTRIBUTE FUNCTION, ATTRIBUTE SEQUENCE
AGGREGATE	=	COUNT(CLAUSE*)
	=	Σ(ATTRIBUTE FUNCTION)
	=	ΣATTRIBUTE FUNCTION; CLAUSE*)
ATTRIBUTE	=	VARIABLE. DOMAIN NAME
Σ	=	COUNT
	=	COUNT'
	=	SUM
	=	SUM'
	=	AVG
	=	AVG'
	=	MAX
	=	MIN

CLAUSE*	=	ATOMIC FORMULA*
	=	ATOMIC FORMULA* $\wedge$ CLAUSE*
ATOMIC FORMULA*	=	(ATTRIBUTE FUNCTION * ATTRIBUTE FUNCTION)
	=	~ ATOMIC FORMULA*

Terminal Symbols

KEY WORDS	RANGE, ALIAS, DESIGNATE, RETRIEVE
NAMES	
CONSTANTS	
VARIABLES	
LOGICAL CONNECTIVES	$\vee$, $\wedge$, ~
COMPARISON OPERATORS	=, >, $\geq$, (>,>0), ($\geq$,>0), (>,$\geq$0)$\cdot$($\geq$,$\geq$0)
MAPPINGS	f^g, $\log_f g$
ARITHMETICAL OPERATORS	+, -, *, /
AGGREGATION OPERATORS	COUNT, COUNT', SUM, SUM', AVG, AVG', MAX, MIN
PUNCTUATIONS	: ; , .
PARENTHESIS	( )
GROUND SYMBOL	Θ

APPENDIX B - PARSING

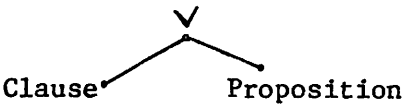
A RETRIEVE query is already a string containing only terminal symbols except for A-functions and proposition. First, we shall reduce a proposition to a string containing terminal symbols and A-functions, then each A-function to a string containing only terminal symbols.

a. Proposition

We shall construct a binary tree as follows:

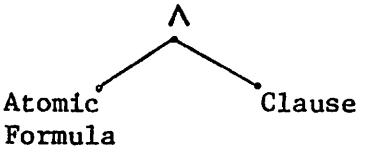
Proposition = Clause

or

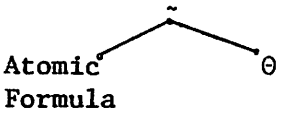


Clause = Atomic Formula

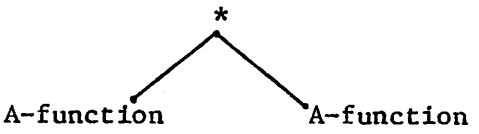
or



Atomic Formula =



or



The binary tree so constructed will have leaves that are either ground symbol Θ or A-functions. The nodes of the binary tree will be arranged in postorder.

b. A-Function

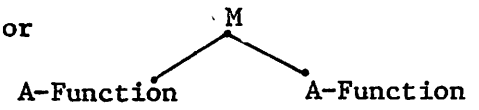
A-Function = Constant

or Aggregate

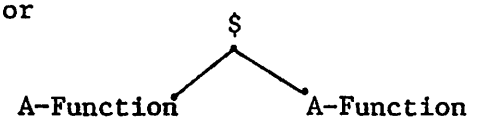
or Attribute Function

or Aggregate Function

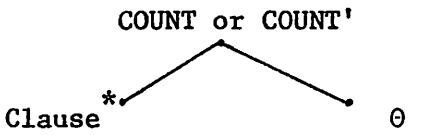
or



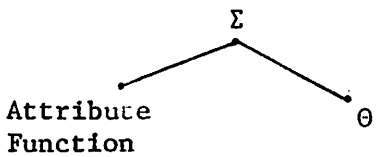
or



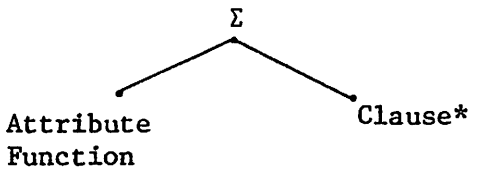
Aggregate =



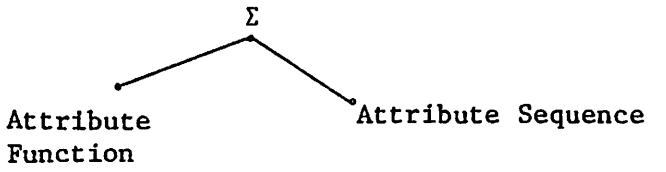
or



or

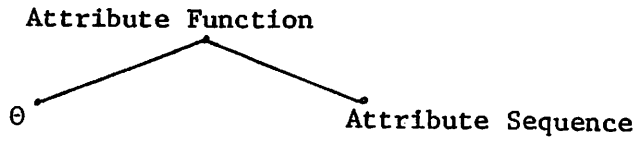


Aggregate = Function



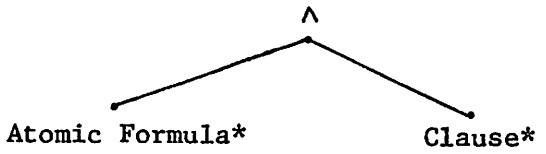
Attribute Sequence = Attribute Function

or

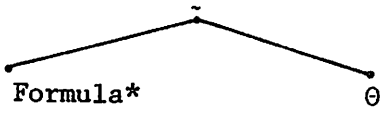


Clause* = Atomic Formula

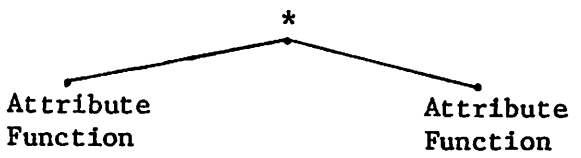
or



Atomic Formula* =



or



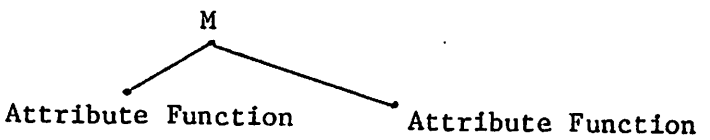
Thus, each A-Function is represented by a binary tree in which the leaves are Θ, and attribute functions. The nodes can put in postorder to yield a string of primitive symobls and attribute functions.

c. Attribute Functions

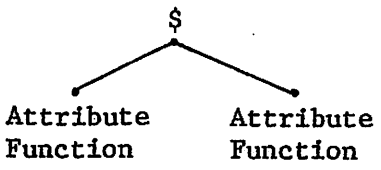
Attribute Function = Constant

or attribute

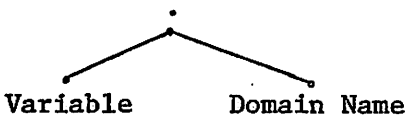
or



or.



Attribute =



APPENDIX C - COMMANDS IN INGRES

Before a discussion of the data management commands can commence we need to discuss the file structure being implemented.

C.2 File Structure

Figure C.1 indicates the proposed directory and file structure. Before a discussion of each individual file, the readers are reminded of the following assumptions.

1) In order to provide any semblance of controlled sharing, INGRES must run in special execute mode.
2) Most INGRES files will have protection status: owner read, owner write, no other access.
3) As a result of 2), there will be one super user (here called XXX) who is the only user allowed to access INGRES files outside of INGRES. This user will sometimes be called DATA CZAR.
4) INGRES will allow several data bases and each is to be protected independently. In Figure C.1 two data bases, LAND and COLLEGE, are indicated.
5) File names in UNIX are the concatenation of the names along the path to a file from the root. Hence, the object code for INGRES is in the file/XXX.INGRES.
6) UNIX file names are 8 alphanumeric characters, the first of which is alphabetic.
7) The same file can have several path names. Hence, sharing of the various files indicated in Figure C.1 is allowed.

We now discuss each file in Figure C.1 in turn.

INGRES - contains object code for our management system. It has special execute protection status. This code will be loaded into the virtual machine of each user of our system.

DATADIR - directory file. Contains pointers to all data bases known to INGRES.

LAND - directory file. Contains pointers to the 3 directories and 3 real files which exist for each data base (in this case, LAND).

WISDOM - This file contains descriptive information about each relation in the data base

HASH - This file contains the hash transformation parameters used to access any hashed relation.

REALITY - This file contains format information on all real relations in the given data base.

REALDIR - directory file. This file contains pointers to all data base files. We adhere to the convention that a relation is stored in a file with the name of the relation's unique id (to be presently discussed).

CODE - directory file. This file contains pointers to all files that serve to code the values of some attribute in the given data base.

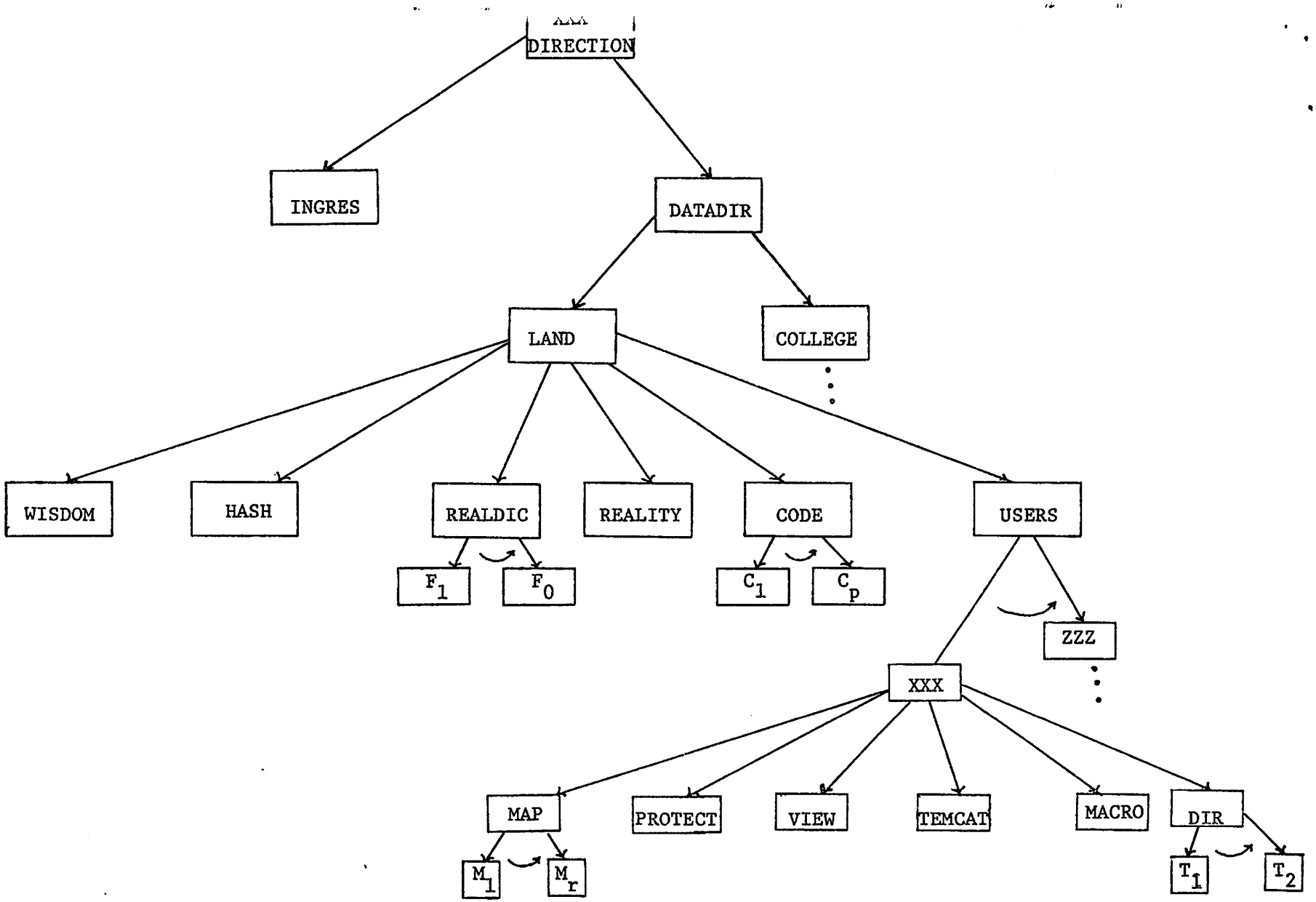


Figure C.1 The INGRES File Structure

USERS - directory file. There is one entry for each authorized user.

YYY - directory file. This file contains pointers to the 4 files and 2 directories unique to a user (here, YYY) of a data base.

PROTECT - This file contains protection information limiting the "view" of the given user.

VIEW - This file indicates the "view" that the given user has of the data base. For the DATA BASE ADMINISTRATOR REALITY $\triangleq$ VIEW.

TEMCAT - This file contains format information concerning temporary files (workspaces).

MACRO - This file contains derived relations (or MACROS) for use by the current user. Many relations in VIEW will have their derivation listed here.

DIR - directory file. This file contains pointers to all temporary files for the user.

MAP - directory file. This file contains pointers to all maps saved by the user.

We will now turn to a detailed discussion of the formats of REALITY, TEMCAT and VIEW.

Formats

REALITY, TEMCAT and VIEW are coded identically and as indicated in Figure C.2. A description of each field now follows.

header - 18 bytes

SPARES - 8 bytes of unused space for future needs

NUMS - (FORMAT C8). This field contains the first unused unique ID for this data base. The first character is a Z; the rest are alphanumeric. Any process needing a unique ID reads this field then increments it by 1. Consequently, at least 36**7 unique file names can be generated starting with Z0000000. This sequence will last for several years. This field is not defined for TEMCAT or VIEW.

OWNER - (FORMAT C8). This field contains the UNIX ID of the owner of the relations cataloged. The owner of the files in REALITY is defined to be the DATA BASE ADMINISTRATOR for the given data base. This user has less power than the DATA CZAR but more than other users.

relation header - 64 bytes

UNIQ ID - (FORMAT C8). This field contains the unique file name where the relation is stored.

REL #1 NAME - This field contains any alias that a user may assign to this relation. (FORMAT C16)

INDEX - (FORMAT C8). A relation and any index for it are chained together on a one-way circularly linked list. This field contains the next member of this chain (in particular its

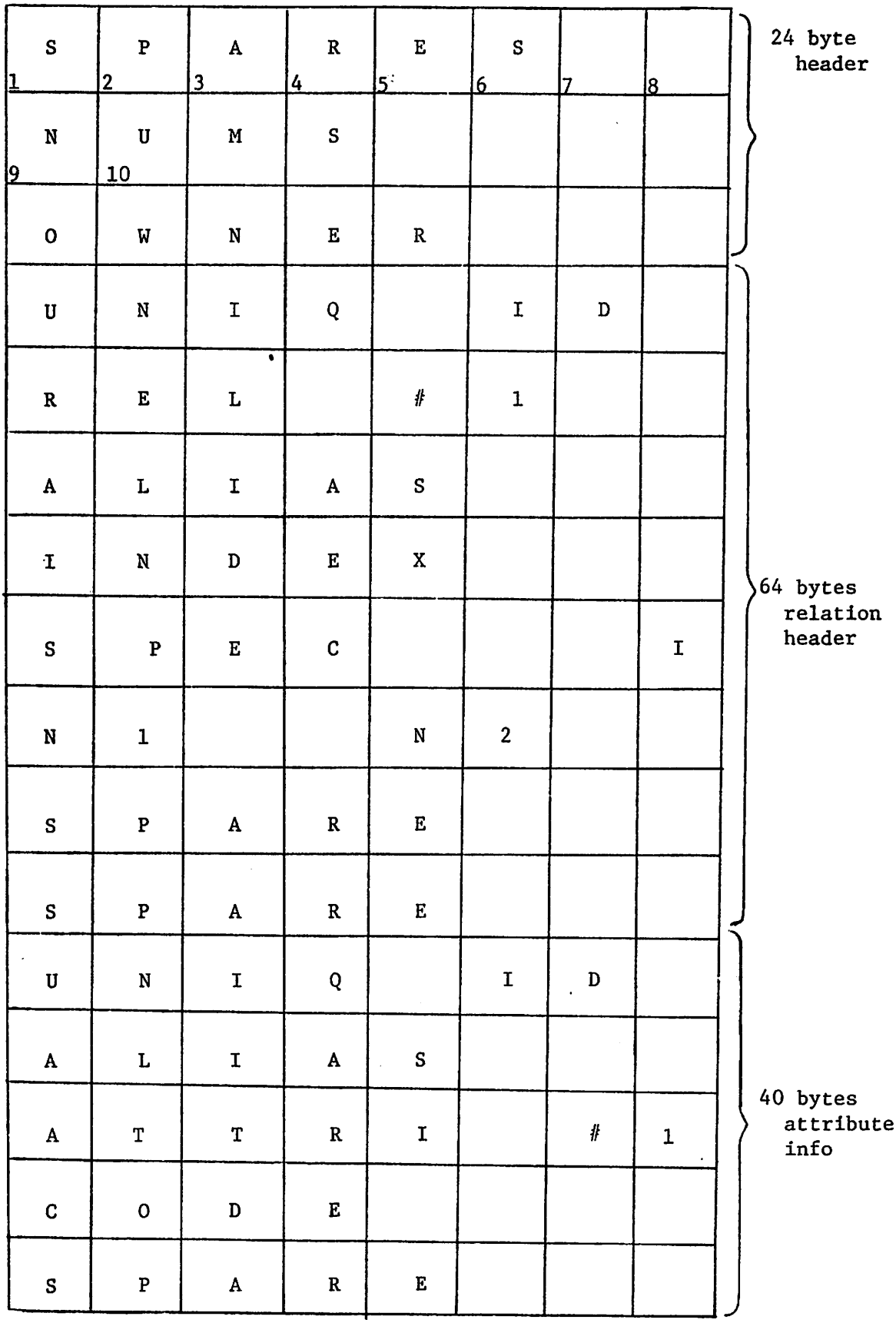

S P A R E S
1 2 3 4 5 6 7 8
N U M S
9 10
O W N E R
U N I Q I D
R E L # 1
A L I A S
I N D E X
S P E C I
N 1 N 2
S P A R E
S P A R E
U N I Q I D
A L I A S
A T T R I # 1
C O D E
S P A R E
24 byte header
64 bytes relation header
40 bytes attribute info

Figure 2.

relation ID). This field is not defined for VIEW.

SPEC - (FORMAT C7). This field indicates how the relation is stored. In particular:

$\not\emptyset$ = relation stored as unsorted table

sort = relation stored as a sorted table

comp = relation sorted and compressed

list = relation stored logically sorted in a list

$NN = relation stored as a hash table. NN indicates how many bytes are to be concatenated and used as a hash key (FORMAT I).

I - Indicates whether this relation is an index or not (FORMAT C1)

0 = relation is an index (follow chain to find out for what)

$\neq$ 0 = this relation is not an index

N1 - (FORMAT 2I). This field contains the number of attributes in the current relation.

N2 - (FORMAT 2I). This field contains the number of tuples in the current relation. This is not defined for VIEW.

attribute information - 40 bytes per attribute

ATTRI #1 ALIAS - This field contains any alias that the user may assign to an attribute. (FORMAT C 16).

UNIQ ID - (FORMAT C8). This field contains the unique id (unique within the realtion) for the current attribute.

CODE - (FORMAT C8). This indicates how the attribute is coded:

SQQQQQQQ = attribute is tree coded with a 2 byte code (2^{16} possible values)

LQQQQQQQ = attribute is tree coded with a 4 byte code (2^{32} possible values)

Here QQQQQQQ is the last 7 bytes of the file name in which codes are to be looked up. Of course, the first byte is "Z"

C(L) = character string of length L

F = 4 byte floating point number

2F = 8 byte floating point number

I = 2 byte integer

2I = 4 byte integer (not yet supported)

It is likely that other codes will be added later.

C.2 User Commands

1. HELP (data base)

In WISDOM helpful textual material is stored concerning the data base opened. HELP is to print out this helpful information as an aid to the beginning user.

2. VIEW

This command empties user VIEW and replaces it with the view allowed by PROTECT. Aliases are obtained from REALITY.

3. RELEASE (relation ID or alias)

This command changes the owner of the indicated relation to that of the current user. If the current user is not the DATA BASE ADMINISTRATOR, only workspaces in the current user's TEMCAT can be released. The DATA BASE ADMINISTRATOR can release a relation in any users TEMCAT or in REALITY.

4. INDEX (Relation id or alias, SPEC, attribute id or alias, ... , attribute id or alias)

This command creates an index for the indicated attributes of the specified relation and catalogs it. The storage mechanism is that of SPEC and defaults to a sorted table. Only the DATA BASE ADMINISTRATOR can use this command.

5. CREATE (SPEC, attribute alias, code, ..., attribute alias, code)

This routine creates an empty field and catalogs appropriate information in TEMCAT. Returns a unique relation id.

6. LINK (file name, length, SPEC, attribute id, code, ..., attribute ID, code)

This has the same general effect as CREATE in that it catalogs a new relation. In addition, however, it links to DIR the file specified and changes its name to a valid INGRES relation id. This command is the opposite of RELEASE. Moreover, the file linked must be in the correct format for the appropriate access method.

7. MODIFY (mode, relation id or alias, SPEC, attribute id, code, ..., attribute id, code)

Mode = 1 ⇒ catalog the relation into REALITY from TEMCAT. This mode is allowed only to the DATA BASE ADMINISTRATOR

Mode = 2 ⇒ recatalog the relation in TEMCAT .

Mode = 3 ⇒ recatalog the relation in REALITY. This mode is allowed only to the DATA BASE ADMINISTRATOR

Mode = 4 recatalog in VIEW

The appropriate catalog entries are changed for the relation specified. The catalog entries for each attribute specified are changed to reflect

a new code indicated. If a "drop" is indicated in code the attribute is deleted. If Mode $\neq 4$ the storage structure of the affected relation is changed to conform to the new catalog entry.

8. PERMUTE (relation id or alias, attribute id or alias, ..., attribute id or alias)

This routine permutes the attributes in a relation usually in preparation for sorting in a specific order.

Again, only the DATA BASE ADMINISTRATOR can perform this operation for relations not in TEMCAT.

9. SORT (relation id or alias)

This routine sorts the indicated relation into collating sequence. It is only defined if the relation is a list or a table. The same restrictions apply concerning authorization to execute this command as for PERMUTE.

10. DELETE (relation ID or alias)

This command destroys a relation. The same restrictions as in PERMUTE apply.

11. ALIAS (relation id or alias, attribute id or alias, new alias)

This command creates a new alias for a given attribute. If an attribute is not specified the new alias is applied to the relations specified.

The following two commands are available only to the DATA BASE ADMINISTRATOR.

12. PROTECT (user, protection statement)

This command inserts a protection statement in a given users PROTECT file

13. DPROTECT (user, protection statement)

This command deletes a protection statement form PROTECT

14. RPROTECT (user, relation)

This command reads and displays the protection statements applied to a given relation and a given user. A user can only display the protection information applied to himself. The DATA BASE ADMINISTRATOR can read everyones protection information.

15.. PROMPT (relation id or alias or blank)

If X = ∅ this routine is to display the user aliases of all relations in TEMCAT and VIEW.

If X ≠ ∅ this routine is to display user aliases of all attributes in the relation with user alias or actual ID of X.

16. FORMAT (relation id or alias)

This command displays all format information concerning a relation in TEMCAT or VIEW

17. CREATE DATA BASE (name)

This command creates a data base with the current user as DATA BASE ADMINISTRATOR. Only a few users will be allowed to execute this command.

18. DESTROY (data base)

The DATA BASE ADMINISTRATOR can destroy an entire data base using this command.